eXtreme Programming

Práticas para o dia a dia no desenvolvimento ágil de software

HISTÓRIAS DE USUÁRIO
ritmo sustentável TESTE DE ACEITAÇÃO
INTEGRAÇÃO CONTÍNUA TDD METÁFORAS DE SISTEMA
FEEDBACK
projeto simples pequenas ENTREGAS
refatoração EXTREME PROGRAMMING time coeso
simplicidade POSSE COLETIVA
REUNIÃO DIÁRIA EM PÉ respeito
CORAGEM SPIKES COMUNICAÇÃO
JOGO DO PLANEJAMENTO
PROGRAMAÇÃO EM PAR

DANIEL WILDT
DIONATAN MOURA
GUILHERME LACERDA
RAFAEL HELM

Prefácio

Prefácio por Klaus Wuestefeld

Poucos são os momentos na vida em que ficamos sabendo de uma ideia e dizemos, na hora: "Esta ideia vai revolucionar o mundo."

Aconteceu comigo quando vi pela primeira vez o Orkut (que descanse em paz). Caiu um raio na minha cabeça e eu disse: "Este negócio (rede social) vai revolucionar o mundo."

Antes disso, em 1999, quando eu ainda chafurdava na burocracia do CMM e do Unified Process, um estagiário da minha empresa me mandou os primeiros links sobre eXtreme Programming (XP).

A princípio me pareceu bobagem, como "Go Horse". Logo vi, porém, que Kent Beck, o autor de XP, e as pessoas discutindo a metodologia no primeiro wiki do mundo eram sérias e sabiam do que estavam falando. Eram, na verdade, os estudiosos de software mais sérios, embasados e apaixonados que eu já tinha visto.

Para entender o que foi, para mim, ver as páginas de XP no primeiro wiki do mundo, imagine dois raios simultâneos caindo na sua cabeça: senti que tanto o wiki quanto os métodos ágeis iriam revolucionar o conhecimento e o desenvolvimento de software.

Comecei a evangelizar XP pelo Brasil e palestrei em um evento de Java em Porto Alegre, em 2004. Estavam lá o Daniel e o Guilherme, coautores deste livro. Caiu um raio na cabeça deles e eles criaram o grupo XP-RS, que promoveu muitos encontros de discussão sobre o assunto.

Agora, onze anos depois, escrevem este livro, mais necessário que nunca.

Cansei de ver times tentando fazer Scrum e entregando zero software em produção, durante meses, sprint após sprint. Cansei de ver times com quadros de Kanban lindos na parede mas com tempo de entrega (lead-time) na ordem

de meses.

É fácil, hoje, fazer um cursinho de final de semana qualquer, certificar-se em charlatanismo ágil de software e sair ditando regrinhas de cerimônias que a equipe de desenvolvimento deve seguir.

Modelos de gestão como Scrum ou Kanban (prefiro) são úteis para ajudar a escolher as histórias certas a desenvolver mas, no fim das contas, o que faz essas histórias sair do outro lado como software rodando sem bugs em produção é a capacidade e dedicação da equipe, usando boas práticas de artesania de software, como as deste livro.

Sugiro às equipes que ganhem fluência, que dominem XP "by the book", antes de inventar moda. O time mais ágil, que produziu o software de maior valor, no prazo mais apertado, nas condições mais politicamente adversas, do qual já participei, foi um time onde aplicávamos XP à risca. Pense muito bem antes de decidir que não precisa de determinada prática.

Quando testar recursos caros ou complicados, utilize constantes que simulem o comportamento desses recursos (com estratégias de dummy, stub, spy, fake e mock).

Não confunda a brevidade de alguns de seus itens, como essa frase, com falta de importância. Este livro é rico e conciso. Cada parágrafo, se explorado, daria outro livro. Já vi uma equipe de dezenas de programadores atrasar em meses seu projeto, por insistir em depender de ambientes de homologação complexos em vez de seguir a dica acima.

Outro exemplo: já vi uma equipe de uma empresa de software para o mercado financeiro desistir de programação em pares sem saber por que. Está num breve parágrafo deste livro o motivo:

Cuide do espaço físico, a mobília pode dificultar ficar lado a lado programando (por exemplo em mesas de canto ou em formato de 'L'). Os programadores terão dores no corpo depois de um dia de programação em pares com pouca ergonomia. Uma dica é adaptar o espaço físico...

Por onde começar? É preciso adotar todas as práticas de uma vez?

Se tiver que escolher alguma prática pra começar, escolha o teste automatizado. Testar software manualmente, no século 21, é antiético, como um cirurgião operar sem lavar as mãos. Aprenda os conceitos, a essência da automação de teste. Não perca tempo tentando controlar telas pra fazer testes.

Separe da tela todo código de negócios e teste-o diretamente.

Programação em pares junto com revezamento são práticas que encontram resistência tanto dos gestores retranqueiros quanto de alguns membros da equipe mais tímidos (respeitar) ou acomodados (incentivar) mas não dependem de mais nada para ser implementadas. Já vi pareamento e revezamento transformar uma equipe desmotivada, à beira de agressões físicas, em uma equipe de referência dentro de sua organização.

Por fim, é vital que a equipe e seu gestor, se houver, reservem um tempo em torno de um dia (projeto novo) a dois dias (projeto com legado macarrônico) por semana com autonomia para a equipe investir em melhorar continuamente sua própria produtividade, aprendendo técnicas como TDD e refatorando código. Se não houver esse tempo alocado e a equipe ficar só correndo atrás de incêndios de curto prazo, é melhor largar o livro agora porque ele só vai trazer frustração.

Se, por outro lado, houver tempo para melhoria, leve este livro como guia, torça para cair um raio na sua cabeça também e boa sorte no caminho.

Aprenda as práticas de XP, domine-as, entregue software do caralho e, aí sim, adapte-as e transcenda.

— Klaus Wuestefeld, pioneiro em eXtreme Programming no Brasil e Keynote Speaker do Agile Brazil (2010)

Prefácio por Alexandre Freire

Se eu tivesse este livro que você tem em mãos na virada do milênio, e conhecesse a Programação eXtrema na época, provavelmente hoje seria milionário, porque teria lançado com sucesso a primeira rede social do mundo.

Estava trabalhando em uma *software house* multinacional, baseada em Treviso, na Itália, construindo uma rede social de times de futebol de várzea para a Diadora. Mas fazíamos isso tudo usando o "melhor", e mais difundido, processo de desenvolvimento de software, o RUP.

Isso significa que durante os 3 anos em que gastei meu suor e sangue, construindo o incrível software desta rede social, nunca tivemos nenhum usuário, além da sócia que fazia demonstrações trimestrais para os diretores da Diadora. E quando toda a diretoria mudou, e a empresa resolveu cancelar o projeto, foi fácil fazê-lo, pois não tinha sido lançado.

Para mim foi uma experiência muito frustrante, afinal com minha equipe tinha construído um software maravilhoso, uma rede social onde times de futebol de bairro poderiam criar uma página, postar fotos, conseguir "likes" de seus fãs e até organizar campeonatos. E o software funcionava que era uma beleza, ou pelo menos funcionava na minha máquina.

Era o maior projeto da empresa, que acabou indo à falência quando perdeu esse cliente.

Deparei-me com a pergunta: *Será que não existe um jeito melhor de fazer software? De entregar mais valor, mais rápido, para os nossos clientes?*. Comecei as estudar o que na época eram as metodologias "leves" e resolvi voltar para o Brasil e para a faculdade. No ano seguinte, participei da primeira turma do Laboratório de Programação eXtrema do IME/USP.

E aí que minha vida de desenvolvedor de software mudou. Aprender e praticar Programação eXtrema foi um marco muito importante na minha carreira, e espero que com este livro, seja um marco na sua também.

Durante muitos anos após esse primeiro encontro, pratiquei a Programação eXtrema como definida nos livros do Kent Beck. Com os erros e acertos, comecei a entender como a metodologia era completa, e como cada parte era preciosa. Também aprendi a fazer algumas coisas bem difíceis muito bem, como estimar em pontos, só para descobrir no final que não precisava realmente fazer isso!

Mas na sua jornada, recomendo que não pule essa prática no começo, a sinergia entre as práticas de XP e o fato de que seus valores e princípios suportam todo o conjunto são umas das razões pelas quais me apaixonei por essa metodologia.

Hoje em dia, a comunidade ágil é forte, tem presença marcante no Brasil, e somos ainda mais extremos. Pegamos algumas práticas e nos desafiamos levá-las ainda mais ao limite! Com as *histórias de usuário* e o *jogo do planejamento* evoluímos pra criar também o *mapa de histórias*. Com *integração contínua* partimos para fazer *deploy contínuo*. Com programação em par radicalizamos para fazer *programação em multirão*.

Mas algumas coisas permanecem para sempre valiosas, como as Retrospectivas. A melhor técnica que conheço para uma equipe se aprimorar continuamente. É tão universal que até já usei em outros contextos que não o

desenvolvimento de software.

Se você for aprender só uma das metodologias ágeis, recomendo fortemente que seja a Programação eXtrema. Mas experimente-a completa, percebendo como os princípios e valores apoiam as práticas, e que estas muitas vezes só funcionam porque dependem das outras. É uma metodologia holística, onde a sinergia conta muito, e com certeza vai melhorar a maneira como você desenvolve software.

Você tem em mãos um guia definitivo para começar sua jornada, aproveite a leitura.

— Alexandre Freire, Diretor de Produtos da Industrial Logic e Keynote do Agile Brazil 2014.

Prefácio por Paulo Caroli

Desde 2000, tenho utilizado (e abusado) de metodologias ágeis. A minha primeira sessão em uma conferência internacional foi na OOPSLA 2000, em que apresentei o resultado da minha tese de mestrado, algo sobre Orientação e Objetos e Design Patterns. Após minha apresentação, fui fazer tietagem com Kent Beck por admirá-lo nos assuntos de smalltalk e design patterns, e acabei comprando seu novo livro com o título *eXtreme Programming Explained*. Deu-se início um caminho sem volta – estava exposto e infectado pela primeira de muitas metodologias ágeis.

Naquela época eu morava no Vale do Silício e estava trabalhando em uma start-up bem legal. Voltei empolgado da conferencia e logo mostrei o livro para o meu gerente da época. Ele riu e disse "Extreme programming... please do not get extreme on our codebase. Be aware of our clients and deadlines". O livro ficou no meu cubículo (naquela época até start-ups tinham cubículos), e de tempos em tempos eu o mostrava para algum colega de trabalho.

Testes unitários e solitários (somente eu os lia e usava). Integração contínua com meu próprio código (os outros desenvolvedores estavam em suas branches). E assim eram meus dias de trabalho em 2000 e 2001.

Mas eu não estava sozinho. Além de Kent Beck, outros conhecidos estavam falando e escrevendo sobre suas experiências com XP. Ward Cunningham populava a C2wiki (http://c2.com/cgi/wiki?ExtremeProgrammingRoadmap) , enquanto Martin Fowler compartilhava

no seu site (http://martinfowler.com/articles/xp2000.html) . Mais desenvolvedores eram infectados, e compartilhavam suas experiências. E isso se alastrou até os dias de hoje, época em que XP é ensinado em faculdades, e muitas de suas práticas são visíveis em todos departamentos de TI.

Em 2006, entrei na ThoughtWorks e trabalhei com pessoas que participaram do início deste movimento ágil. Por exemplo, ouvi as histórias do Martin sobre o primeiro projeto XP, o C3 (Chrysler Comprehensive Compensation). Vivi e ouvi relatos de experiência sobre a implantações de XP ao redor do mundo. Em 2008 e 2009 constatei que até projetos de outsourcing na Índia estavam usando XP. Bah, mas eu seguia referenciando aquele livro de 2000!

Em 2010, eu ajudei a trazer a ThoughtWorks para o Brasil, abrindo seu primeiro escritório em Porto Alegre. A escolha da cidade não foi somente pelo famoso churrasco, mas sim por admiração ao movimento de XP, e a excelente comunidade já existente. Conheci e aprendi com cada um dos autores deste livro. Daniel, Dionatan, Rafael e Guilherme estão sempre compartilhando conhecimento. Nesta obra, compartilham os valores, práticas e princípios de XP.

Com muito orgulho, informo que este livro é um resumo do que há de melhor no mundo atual de desenvolvimento de software.

Boa leitura,

— Paulo Caroli, cofundador e agile coach da Thoughtworks Brasil.

Agradecimentos

Antes de tudo, gostaríamos de agradecer a toda a comunidade ágil brasileira, que é uma das maiores e mais participativas no mundo. Temos grandes eventos em nossa área de métodos ágeis com agilistas muito experientes e somos gratos por isso. Obrigado por todas as trocas de experiências durante esses anos de agilidade no Brasil!

À comunidade do antigo XP-RS, o atual GUMA-RS Grupo de Usuários de Métodos Ágeis do Rio Grande do Sul (saiba mais em http://www.guma-rs.org) , o nosso muito obrigado por ser um grande grupo fomentador da agilidade e do eXtreme Programming!

Obrigado, e muito, aos nossos *beta readers* Jamile Alves, Maurício Aniche e Rodrigo Pinho! As opiniões e sugestões de vocês foram essenciais para a melhoria do livro.

Obrigado também a todas as pessoas da Casa do Código. Vocês nos deram um excelente suporte e sempre estiveram dispostos a contribuir com esta obra!

Aos agilistas Klaus Wuestefeld, Alexandre Freire e Paulo Caroli, a nossa gratidão pelos admiráveis prefácios!

A nossos alunos e clientes, que nos permitem praticar e aprender em conjunto, e a você leitor, o nosso muito obrigado por interessar-se neste trabalho! Esperamos que a leitura deste livro seja útil e agradável! :-)

Dedico este livro aos desenvolvedores profissionais do Brasil, que se preocupam em entregar software de qualidade e buscam excelência técnica! — Daniel Wildt

Agradeço a todos que veem nos métodos ágeis um caminho para um mundo melhor! Dedico este livro aos meus pais e irmãos! — Dionatan Moura

Dedico à minha família: em especial, para meus pais (Paulo e Liberta) e para minha esposa e filha (Juliana e Isabella)! — Guilherme Lacerda

Agradeço à Casa do Código pela oportunidade, ao Klaus Wuestefeld, Alexandre Freire e Paulo Caroli pelos prefácios inspiradores, à Vivian pela dedicação e paciência durante a revisão deste livro, aos meus amigos e parceiros de jornada, Daniel, Dionatan e Guilherme! Além é claro da minha esposa Aline e meu filho Rafael Helm Junior, com eles ao meu lado tudo faz sentido. — Rafael Helm

Sobre os autores

Daniel Wildt

Fundador em 2004 do XP-RS Grupo de Usuários de eXtreme Programming do RS , que depois se transformou no GUMA-RS Grupo de Usuários de Métodos Ágeis do Rio Grande do Sul. Desde 2003, um praticante interessado pelo desenvolvimento de comunidades, organizações, equipes e pessoas em torno da agilidade. Trabalha em processos de melhoria das técnicas de engenharia de software e processos, sempre em busca de simplicidade e do aprendizado. Ajuda pessoas a tornarem-se melhores profissionais, equipes a crescerem por meio de melhoria contínua e produtos/serviços a serem formados. Além da formação técnica, possui formação de Master Trainer em Programação Neurolinguística. É CTO na uMov.me (http://umov.me) , coach/instrutor pela Wildtech (http://www.wildtech.com.br) e facilitador no Estaleiro Liberdade (http://www.estaleiroliberdade.com.br) .

Dionatan Moura

Coach ágil na Companhia de Processamento Dados do Estado Rio Grande Sul (PROCERGS), auxiliando na entrega de maior valor ao cidadão por meio da agilidade e inovação. Professor visitante na pós-graduação da UniRitter Laureate International Universities. Mestre e bacharel em Ciência da Computação na UFRGS. Coordenador do Grupo de Usuário de Métodos Ageis do RS (GUMA-RS) e do Grupo de Usuários Java do RS (RSJUG). Foi desenvolvedor Java por sete anos, e coordenador das trilhas Java do TDC POA 2013 e 2014. Palestrante em eventos ágeis, como: Agile Brazil, TDC e XPConf; e possui as certificações: CSP, SAFe SA, PMP, CSM, PSM I, CSD, CSPO, OCPJP, CTFL, ITIL e MPS-BR. Pode ser encontrado no Twitter como @dionatanmoura e possui um blog: http://dionatanmoura.com.

Guilherme Lacerda

Pioneiro em Metodologias Ágeis no Brasil, no qual atua desde 2001, com especial ênfase em Lean, SCRUM e eXtreme Programming. Trabalha como Coach, ajudando na adoção de métodos ágeis no Tribunal de Justiça do RS. Professor universitário de graduação (UniRitter) e de pós-graduação (UniRitter, Unisinos, UFRGS). Mestre em Ciência da Computação (UFRGS, 2005), área de Engenharia de Software e atualmente doutorando pela mesma instituição, com estudos na área de Refatoração e Smells. Consultor associado da Wildtech, palestrante em dezenas de eventos nacionais e internacionais sobre o tema, e fundador e vice-coordenador do XP-RS/GUMA. Possui as certificações CSM e CSP, pela Scrum Alliance. Membro do IASA, ScrumAlliance, ACM e SBC. Possui os blogs: http://www.guilhermelacerda.net e http://www.codingbyexample.org.

Rafael Helm

Sócio da Wildtech (www.wildtech.com.br), na qual atua como agile coach, instrutor e consultor. Experiência de agile coach adquirida em projetos de larga escala em organizações como Receita Federal, Procuradoria Geral da Fazenda Nacional, Ministério de Relações Exteriores, entre outras. Instrutor na Wildtech, tendo a oportunidade de treinar empresas e equipes de diferentes localidades do Brasil, incluindo o Ministério da Justiça, DNIT, PROCERGS, Serpro, Totvs, entre outras. Palestrante em eventos importantes como Agile Brazil, The Developers Conference, FISL, XP Conf BR. Autor do e-book *Histórias de Usuário - Por que e como escrever requisitos de forma ágil?*, que atingiu a marca de 4.000 downloads em menos de um ano (www.wildtech.com.br/historias-de-usuario). Idealizador e cofundador da XP Conf. BR, a conferência nacional sobre eXtreme Programming, e cofundador das conferências UX Conf. BR, DevOps Conf. BR e Empreende Conf. https://about.me/rafaelhelm

Você pode participar da lista de discussão deste livro! Basta escrever para: livro-xp@googlegroups.com.

Sumário

Capítulo 1

Por que projetos falham?

Culpa do cliente, do gerente de projetos, dos testadores, dos analistas de negócios, do time de desenvolvimento, ou simplesmente falta de sorte? Afinal de contas, por que projetos falham, atrasam e/ou fracassam?

Os desenvolvedores acham que faltou análise; os analistas jogam a culpa no departamento comercial que vendeu o que o software não tem e, ainda por cima, com um prazo impossível de ser cumprido; a equipe de vendas aponta para o cliente dizendo que ele não sabe o que quer; e ele, por sua vez, chama seus advogados para cancelar o contrato.

É bem provável que você já tenha passado por isso, ou, pelo menos, quase passou.

Infelizmente, nós, os autores deste livro, já vimos isso acontecer algumas vezes, e normalmente os motivos são os mesmos: falta de proximidade com o cliente; trabalho empurrado goela abaixo do time; codificação a toque de

caixa produzindo código sujo; tempo desperdiçado desenvolvendo o que o cliente não pediu (a famosa cereja do bolo); e o retrógado hábito de testar o software apenas na véspera da sua data de entrega.

Sendo assim, achamos importante começarmos este livro reforçando por que você não deve cometer estes erros clássicos em seus projetos, para, somente depois, entrar no conteúdo do *eXtreme Programming* propriamente dito. Leia com atenção e lembre-se: antes de prescrever o remédio, é necessário identificar a doença.

1.1 Cliente distante

Cena corriqueira em times de desenvolvimento de software: o desenvolvedor recebe uma especificação para trabalhar, e, enquanto está codificando, percebe que algum detalhe passou despercebido pelo processo de análise do requisito. Isto gerou uma dúvida que o impede de prosseguir seu trabalho com segurança; ou seja, o desenvolvedor não tem certeza do que exatamente deve ser feito.

Nesses casos de incerteza, normalmente ele procura o analista ou a pessoa responsável por aquela especificação (ou pelo menos deveria fazer isso), e questiona sobre a dúvida que o impede de avançar no código com a certeza de que está fazendo o que foi pedido.

Quando isso ocorre, é esperado que o analista saiba responder os questionamentos do desenvolvedor e o conduza de volta ao caminho correto a ser seguido. O problema é que as pessoas não sabem tudo, incluindo eu, você, e todos os que nós conhecemos. É normal que, em alguns casos, o analista não consiga responder a dúvida do desenvolvedor, pelo menos, não de imediato.

Quando as dúvidas surgem e ninguém do time sabe o que fazer, a melhor decisão é falar com o cliente, seja por meio de uma reunião, telefone, skype, e-mail ou qualquer outra forma de comunicação. O necessário é falar com ele!

Entrar no ciclo *tentativa/ erro/ tentativa/ erro/ ...* jamais deve ser uma opção. As pessoas até podem acertar de vez em quando, mas, na média, essa atitude vai gerar retrabalho lá na frente, além de frustrar o cliente e, posteriormente, o time.

Sabemos que nem sempre ele estará disponível. Mesmo que esteja fisicamente próximo de você e de seu time, ele pode simplesmente não estar livre ou interessado em ajudar; quer dizer, não quer se envolver. Até porque já pagou uma boa grana para uma empresa fazer seu software e espera que o time de desenvolvimento saiba como fazê-lo.

Pois bem, este cenário é o que chamamos de **cliente distante**. Acredite, projetos com clientes distantes têm uma grande tendência a serem entregues com atraso, estourarem o orçamento, os frustrarem ao entregar funcionalidades que não atendam suas expectativas, ou todas as alternativas anteriores juntas (fracasso tremendo).

E qual é a solução?

Precisamos aproximar o cliente desde o início do projeto, deixando bem claro que, a qualquer momento, ele poderá ser acionado para tirar dúvidas, e que a cada funcionalidade, ele poderá ser chamado para realizar uma validação. Ou seja, dar seu consentimento sobre se aquele recurso ficou de acordo com o que esperava, ou indicar o que precisa ser ajustado para atendê-lo, caso o resultado não satisfaça suas expectativas.

Porém, é necessário tomar cuidado. Pedir o aceite do cliente não significa pedir para ele testar. O teste do software é responsabilidade do time de desenvolvimento e, acredite em nós, eles não gostam de testar software. E, principalmente, não ficam nada felizes quando encontram *bugs* nele.

Quando pedimos para ele validar ou aceitar uma funcionalidade, na verdade estamos buscando um *feedback*. Queremos confirmar se aquela função que acabou de sair do forno está de acordo com a sua expectativa. Estamos buscando segurança para continuar o desenvolvimento do projeto, tendo certeza de que os passos dados até agora foram realizados na direção correta.

E se o cliente odiar a funcionalidade desenvolvida, dizendo que está totalmente fora do que ele esperava? Ótimo! Nós conseguimos falhar rápido e teremos tempo para corrigir, adaptar ou até mesmo refazer o recurso, conseguindo finalmente entregar algo de valor a ele. Imagine como seria desagradável se, somente no final do projeto, nós descobríssemos que algumas (ou várias) funcionalidades não ficaram de acordo com sua expectativa? Seria terrível! E provavelmente nosso tempo já estaria esgotado.

Então, aproxime-se de seu cliente e mantenham-se próximos!

1.2 A cereja do bolo

Você tem seus clientes e eles têm expectativas. Se algumas empresas que produzem software criassem uma confeitaria especializada em bolos, muitos clientes receberiam apenas uma receita, ensinando como fazer o *maldito* bolo. Digamos que isso não gera nem tanto valor, nem satisfação. Ele estava com fome, queria *só* comer um bolo.

Outros clientes com um pouco mais de sorte receberiam uma massa, nem tão bem batida, mas que faria um belo bolo "abatumado". Depois de perder vários prazos com o mesmo cliente (o qual, por alguma magia, ainda segue comprando bolo daquele local), acontece que, eventualmente, a empresa consegue terminar o bolo, exatamente como ele pediu. Só que, lembra das suas experiências anteriores, nas quais ele não o recebeu? Então, a empresa não pode entregar só um simples bolo. É preciso entregar algo maior, algo a mais. Surpreender!

Não precisamos da cereja do bolo! Ou se você gosta de usar o PMBOK® (*Project Management Body of Knowledge* Corpo de Conhecimento em Gerência de Projetos) como exemplo; estamos falando do *Gold Plating*.

Se o seu cliente solicita um relatório com as colunas A, B e C; o time deve desenvolver, com qualidade e dentro do prazo, o relatório com as colunas A, B e C. Simples assim!

Entregar um relatório com as colunas A, B, C, X e Y não faz sentido nenhum! Se ele quisesse que as colunas X e Y aparecessem no relatório, teria solicitado. Neste caso, X e Y são a tal cereja do bolo da qual estamos falando.

Vamos lembrar de algo bem importante. Nosso cliente tem uma grande expectativa: receber aquilo que ele pediu! E ele possui outras também! Quer receber o mais rápido possível, com a mais alta qualidade e o menor custo. E com o menor desperdício possível, é claro. Colocar uma cereja no bolo não vai ajudá-lo a esquecer as falhas anteriores. Nem nas próximas entregas.

E qual é a solução?

Simplicidade: maximizar o trabalho que não deve ser feito. A meta do time deve ser trabalhar com o cliente para enxugar as funcionalidades necessárias. Deve-se questionar muito sua visão de valor.

Muitos recursos sugeridos por ele não possuem um método fácil de medir o valor ou, até mesmo, o Retorno Sobre Investimento (ROI *Return On Investment*). Uma simples classificação para entender o valor e o risco de fazermos uma determinada funcionalidade pode ser uma boa racionalização para ajudar na seleção do que precisa ser feito.

Indo nessa linha, vamos começar trabalhando naquilo que possui alto risco e valor, e descobrir rapidamente se temos algum problema. Lembra da estratégia de falhar rápido? Esse é o jogo. O mundo perfeito é trabalhar naquilo que possui alto valor e um baixo risco!

Não queremos trabalhar naquilo que possui baixo valor. O único motivo para isso está relacionado às leis, nas quais temos que seguir para a conformidade e evitar alguma multa. De resto, fuja daquilo que não tem valor claro. Foque na entrega de valor para seu cliente.

1.3 Testes no final

Afinal, por que testamos? Aliás, é necessário realmente testar?. Ou pior: *se sobrar tempo, testamos!*

Essas expressões são comumente ditas por profissionais de software. Infelizmente.

Deixe-nos explicar melhor: desenvolver software é um processo produtivo, no qual executamos tarefas de investigação, design, programação, testes, entre outros. A forma como a indústria começou a desenvolver seus modelos (e independente do modelo a ser adotado) engloba essas etapas.

A questão a ser observada no processo é que a preocupação com os testes só existe no final. Além de acontecer isoladamente, não se enxerga seu verdadeiro valor, nem sua relação com a qualidade do produto.

Por esses motivos, os testes não são levados a sério. Por isso, pagamos o preço do retrabalho ao entregar funcionalidades erradas e com falhas. Isso gera desconfiança e estressa uma relação que precisa ser totalmente diferente para atingirmos o objetivo.

E qual é a solução?

Precisamos trabalhar nos recursos até eles estarem prontos de verdade; ou seja, não podemos permitir que exista dentro do time de desenvolvimento o conceito de que *está pronto, só falta testar*.

O teste não deve ser encarado como uma etapa à parte, como algo extra. Os processos de desenvolvimento e testes devem ser vistos como uma única etapa. Esqueça o *desenvolver primeiro, testar mais tarde*, e busque o *desenvolver + testar*. Entende a diferença? As funcionalidades não são testadas mais tarde; elas são testadas agora.

1.4 Trabalho empurrado

Imagine a cena: o gerente de projetos avisa o time de desenvolvimento de que a empresa fechou um novo contrato e que o software deve ser entregue em uma determinada data, deixando já bem claro a todos que não existe possibilidade de não a cumprir.

O time de desenvolvimento começa a analisar os requisitos e gerar as tarefas, e logo percebe que, para cumprir a data já firmada com o cliente, que inclusive consta no contrato, será necessário produzir em uma velocidade muito acima do habitual. Ao notar isso, o nível de estresse do time é elevado, o ambiente fica tenso e os desenvolvedores programam a toque de caixa, começando a gerar um código sujo, mal estruturado e em cima de requisitos especificados de forma superficial. Como o tempo era curto, ninguém quis *perder* muito tempo com especificação, assim todos já foram logo metendo a mão na massa.

Esse é um claro cenário de trabalho empurrado, que gera muito caos e pouco resultado!

E qual é a solução?

Se trabalho empurrado é um problema, trabalho puxado é a solução. O ideal é que esteja à disposição do time o que for necessário para o desenvolvimento de software e que as tarefas estejam corretamente priorizadas e, se possível, dispostas em um mural contendo todas as que serão trabalhadas

nos próximos dias. Desse modo, uma ou mais pessoas puxam as tarefas e trabalham nelas até o fim, nunca esquecendo de *desenvolver + testar*, em vez de *desenvolver primeiro, testar mais tarde*.

Após terminar a tarefa, o desenvolvedor vai até o mural, dá uma olhada nas próximas da fila e puxa aquela em que ele sente que consegue trabalhar. Claro que é necessário que ele faça isso com responsabilidade; ou seja, é importante que, dentro do possível, ele siga a priorização. A exceção a essa regra pode ocorrer quando a próxima tarefa da fila refere-se, por exemplo, a um módulo que ele não conhece ou a uma tecnologia com qual ele não se sente seguro. Porém, esses casos são ótimas oportunidades para trabalhar em par com outra pessoa do time e aprender algo novo, tanto em termos de negócio, quanto de tecnologia.

Nós acreditamos fortemente no trabalho puxado e sempre orientamos nossos times a trabalharem desta forma, de preferência baseados em um mural que permita gestão visual das tarefas e do progresso de trabalho, seja no meio físico ou virtual (no caso de equipes distribuídas).

Sabemos que quebrar a cultura do trabalho empurrado não é fácil, pois existe (principalmente na cabeça da maioria dos gestores) uma barreira muito forte em relação a isso. Entretanto, garantimos que vale a pena tentar. Durante o livro, vamos mostrar ferramentas e táticas que nos ajudarão nessa missão, como: time coeso, posse coletiva, liberação frequente de pequenas entregas, trabalho em pares, jogo do planejamento, e outros tópicos que ajudam a implantar a cultura de trabalho puxado.

1.5 Dívidas técnicas

O projeto começa e o time de desenvolvimento está trabalhando com alta produtividade. Várias partes do software vão sendo construídas e todos estão felizes.

Passadas algumas semanas, a produtividade caiu um pouco e a euforia do time passou; mas o projeto ainda transcorre dentro da normalidade e eles ainda estão relativamente contentes.

Passadas mais algumas semanas, a produtividade cai drasticamente e muitos bugs começam a ser encontrados. Nesse momento, o time já está com

a moral baixa, o cliente notou que o projeto não anda bem, o custo de mudança de qualquer funcionalidade é muito alto, e você não entende o motivo dessa queda; afinal de contas, o time é bom, a especificação foi bem feita, o cliente está próximo, a comunicação flui e as pessoas são comprometidas. O que está acontecendo?

É bem provável que o projeto possua uma dívida técnica grande.

O termo *dívida técnica* é utilizado para indicar trechos de código que foram mal escritos, ou escritos de qualquer forma, sem refatorar, sem testes automatizados e sem respeitar padrões. São as chamadas *gambiarras* ou *bacas*. Com o passar do tempo, esses trechos atrapalharão os desenvolvedores e mudanças que, teoricamente, seriam simples, passam a levar muito tempo para serem realizadas.

É provável que você já tenha deparado-se com uma funcionalidade praticamente impossível de ser evoluída, tamanha era a falta de qualidade do código. Em alguns casos, quando isso ocorre, somos obrigados a refazê-la toda, pois o código tornou-se ilegível e isso gera um grande desperdício de tempo e dinheiro.

Muitos desenvolvedores ainda consideram que qualidade do código é algo extra e que o foco é entregar software com qualidade, não necessariamente produzindo **código** de qualidade. Essa é uma escolha perigosa, porque, a menos que o projeto seja bem pequeno, o desenvolvedor precisará pagar a conta da dívida técnica produzida, ou simplesmente o projeto vai parar.

E qual é a solução?

São excelentes formas de evitar a dívida técnica: especificar requisitos com o formato de histórias de usuários, projeto simples, padrão de codificação, desenvolvimento guiado por testes (TDD), programação em pares, refatoração, integração contínua, boas práticas de programação e ritmo sustentável.

Mas, se você não sabe como implantar essas práticas no seu time de desenvolvimento, fique tranquilo! Estamos apenas no começo do livro e abordaremos esses assuntos mais à frente.

1.6 Conclusão

Quando um projeto falhar, não tente encontrar um culpado, olhe para o passado e procure identificar se algum dos problemas ocorreu em seu projeto. Precisamos sempre buscar sua causa raiz. Não busque culpados, busque entender o que ocorreu e converse com seu time para tentar encontrar uma solução, um melhor caminho a ser seguido.

Reuniões de retrospectiva de projeto são ótimas oportunidades de levantar problemas e discutir soluções com seu time. Mais à frente, falaremos sobre elas.

Lembre-se: em métodos ágeis, incluindo o eXtreme Programming, devemos sempre trabalhar com a mente aberta para realizar adaptações em nosso processo de trabalho, a fim de buscar melhoria contínua. É uma busca infinita pelo **fazer melhor**.

Capítulo 2

Introdução a métodos ágeis

Em fevereiro de 2001, em Utah, 17 profissionais experientes (consultores, desenvolvedores e líderes) da comunidade de software reuniram-se para discutir alternativas aos processos e às práticas burocráticas utilizadas nas abordagens tradicionais de Engenharia de Software e Gerência de Projetos. Assim nasceu o **Manifesto Ágil** [17], destacando as diferenças em relação às abordagens tradicionais.

A principal diferença das metodologias ágeis em relação às tradicionais é o enfoque na adaptação, visando ter o mínimo realmente necessário para a realização do trabalho. Com essa estratégia, busca-se aceitar e trabalhar a mudança, reduzindo custos de implementação.

2.1 O Manifesto Ágil

O Manifesto Ágil é uma declaração dos valores e dos princípios que fundamentam o desenvolvimento ágil de software. Ele foi assinado em 2001 por dezessete profissionais de software que já vinham experimentando metodologias e práticas mais leves e enxutas. Isso foi tão bem recebido pela comunidade que alguns desses desenvolvedores ganharam o mundo palestrando sobre o assunto. O Manifesto Ágil possui 4 valores e 12 princípios, que vamos apresentar em seguida.

Os 4 valores do Manifesto Ágil são:

- Indivíduos e interações **mais que** processos e ferramentas;
- Software em funcionamento **mais que** documentação abrangente;
- Colaboração com o cliente **mais que** negociação de contratos;
- Responder a mudanças **mais que** seguir um plano.

E os 12 princípios do Manifesto Ágil são:

1) Nossa maior prioridade é satisfazer o cliente por meio da entrega contínua e adiantada de software com valor agregado;

2) Mudanças nos requisitos são bem-vindas no desenvolvimento, mesmo tardiamente. Processos ágeis tiram vantagem das mudanças, visando vantagem competitiva para o cliente;

3) Entregar frequentemente software funcionando, de poucas semanas a poucos meses, com preferência à menor escala de tempo;

4) Pessoas de negócio e desenvolvedores devem trabalhar diariamente em conjunto por todo o projeto;

5) Construa projetos em torno de indivíduos motivados. Dê a eles o ambiente e o suporte necessários e confie neles para fazer o trabalho;

6) O método mais eficiente e eficaz de transmitir informações para (e entre) uma equipe de desenvolvimento é por meio de conversa face a face;

7) Software funcionando é a medida primária de progresso;

8) Os processos ágeis promovem desenvolvimento sustentável. Os patrocinadores, desenvolvedores e usuários devem ser capazes de manter um ritmo constante indefinidamente;

9) Contínua atenção à excelência técnica e bom design aumentam a agilidade;

10) Simplicidade a arte de maximizar a quantidade de trabalho não realizado é essencial;

11) As melhores arquiteturas, requisitos e designs emergem de equipes auto-organizáveis;

12) Em intervalos regulares, a equipe reflete sobre como tornar-se mais eficaz e, então, refina e ajusta seu comportamento de acordo.

Esses princípios, se bem compreendidos pelo cliente e pelo time, promovem uma mudança de atitude. O cliente consegue enxergar valor mais rapidamente nas entregas frequentes do software e, à medida que visualiza a solução, consegue refletir sobre alternativas e prioridades. O time trabalha mais motivado, porque consegue enxergar valor no seu trabalho que, por meio de feedback constante, aprimora continuamente. O bom relacionamento melhora para ambos, visto que a confiança se faz cada vez mais presente.

Com base nesses valores e princípios, aparecem três abordagens mais específicas que vêm proporcionado ótimos resultados, seguindo a essência do Manifesto Ágil. A seguir, elas são apresentadas rapidamente.

2.2 Lean Software Development

O Lean, ou Sistema Toyota de Produção (STP), revolucionou a indústria da manufatura, tanto pela geração de valor para o cliente, quanto para a satisfação de trabalho para o time. Essas suas características serviram de base para o casal Mary e Tom Poppendieck identificar semelhanças com o desenvolvimento ágil de software, criando o *Lean Software Development* (LSD) [50].

Com base nos seus pensamentos, foram identificados 7 princípios para o desenvolvimento de software:

1) Eliminar desperdícios;
2) Incluir qualidade no processo;
3) Criar conhecimento;
4) Adiar comprometimentos;
5) Entregar rápido;
6) Respeitar as pessoas;
7) Otimizar o todo.

O modo Lean de pensar é muito útil para criar uma cultura de melhoria contínua nas equipes. Seus princípios são perfeitamente alinhados aos valores e às práticas do Manifesto Ágil.

2.3 Scrum

O Scrum surgiu na década de 90, criado por Ken Schwaber, Jeff Shuterland e Mike Beedle. Seu principal objetivo é prover um framework de gestão ágil para desenvolver e manter produtos complexos [51].

Nele, a partir de um *Backlog de Produto* inicial, o trabalho que será realizado é priorizado na iteração, denominada *Sprint*. O desenvolvimento do *Backlog da Sprint* gera um incremento entregável do produto ao final de cada *Sprint*, na reunião de *Revisão da Sprint*. Esse trabalho é desenvolvido com a sincronização diária do *Time de Desenvolvimento*, na *Reunião Diária*. Ao final da *Sprint*, é realizada a reunião de *Retrospectiva da Sprint*, com o objetivo de reduzir riscos e promover a melhoria contínua. O *Dono do Produto* é o papel responsável pelo gerenciamento do produto e o *ScrumMaster* é pelo processo, ajudando o *Time de Desenvolvimento* a resolver seus impedimentos.

Scrum precisa das práticas técnicas do XP

Scrum é um framework bastante popular atualmente e muito utilizado para o gerenciamento de projetos, produtos e times. Por dar mais ênfase no gerenciamento de atividades e tarefas, ele pode (e deve) ser utilizado em conjunto com outros métodos e processos que focam em engenharia ágil de software, como o eXtreme Programming.

Jeff Sutherland, cocriador do Scrum, diz que o Manifesto Ágil é sobre Scrum com práticas de engenharia do eXtreme Programming, e essa combinação é o framework preferencial [55].

Veja também o que Ron Jeffries, Martin Fowler e James Shore falam da necessidade das práticas do XP na utilização do Scrum:

"Você não vê times Scrum de alto desempenho sem práticas de engenharia do XP. É difícil escalar times XP sem Scrum, e Scrum soluciona os assuntos de interface com a gestão. Seja cuidadoso ao fazer partes de qualquer coisa e chamar isso de ágil."
– Ron Jeffries e Jeff Sutherland [56]

"Se você estiver procurando introduzir Scrum, certifique-se de prestar uma boa atenção às práticas técnicas. Nós tendemos a aplicar muitas dessas a partir do Extreme Programming e elas se encaixam muito bem."
– Martin Fowler [25]

"XP e Scrum são as melhores maneiras que nós sabemos para trabalhar bem em conjunto. Eles não são necessários pode haver outras maneiras e eles certamente não são suficientes há um milhão de coisas que devemos fazer para ser bem-sucedido, e elas não estão todas escritas nos livros."
– Ron Jeffries [32]

"Sem práticas de engenharia ágil do XP, a qualidade do código e a produtividade diminui assintoticamente ao longo do tempo. Com elas, a produtividade começa inferior, mas em seguida aumenta assintoticamente."
– James Shore [52]

2.4 eXtreme Programming (XP)

O eXtreme Programming é uma metodologia ágil de desenvolvimento de software voltada para times de pequeno a médio porte, no qual os requisitos são vagos e mudam frequentemente. Desenvolvido por Kent Beck, Ward Cunningham e Ron Jeffries [6], o XP tem como principal tarefa a codificação com ênfase menor nos processos formais de desenvolvimento e com uma maior disciplina de engenharia ágil de software para codificação e testes. Tem como valores a comunicação, a simplicidade, o feedback, a coragem e o respeito.

O XP valoriza a automatização de testes, sendo estes criados antes, durante e depois da codificação. É flexível para a mudanças de requisitos, valorizando o feedback com o usuário e a qualidade do código-fonte final.

A ideia principal do XP é a criação de software de alta qualidade, abandonando todo tipo de *overhead* de processo que não suporte diretamente a entrega de valor. Ele é orientado explicitamente às pessoas e vai contra o senso comum do gerenciamento de que elas são peças intercambiáveis dentro do processo de desenvolvimento.

Sinergia entre as práticas

Nos próximos capítulos, detalharemos os valores, os papéis e as diversas práticas do XP, trazendo a teoria e a prática com muitas dicas. Uma prática do XP, um valor ou um papel serve como suporte para outras práticas, outros valores e outros papéis do próprio XP, criando uma sinergia. Por exemplo, as histórias de usuário são base para os testes de aceitação, no qual rodarão na integração contínua e permitirão pequenas entregas, auxiliando no ritmo sustentável e na comunicação do time de desenvolvimento com o cliente. Quanto mais afinados eles estiverem, mais benefícios o XP trará ao desenvolvimento de software.

Esperamos que você navegue de forma prazerosa pelas páginas a seguir. Para auxiliá-lo nisso, disponibilizamos a *big picture* do XP com seus valores e práticas de gestão, planejamento, projeto e codificação (vide imagem a seguir). Você pode baixar e imprimi-la em [2].

Esperamos também que este livro mude sua forma de pensar e agir (se

ainda não mudou), e ajude você, seus clientes e seu time a atingirem satisfatoriamente seus objetivos, de preferência dentro do prazo, custo e orçamento previstos. :)
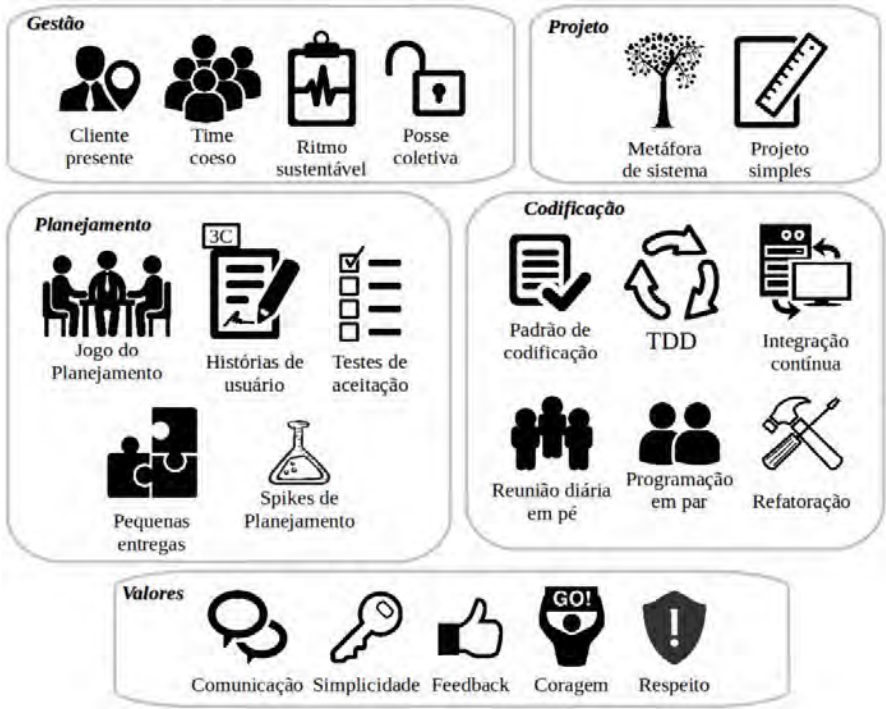
Fig. 2.1: Big picture do XP. Imagens de Freepik (http://www.flaticon.com/authors/freepik) em Flaticon (http://www.flaticon.com) com a licença Creative Commons BY 3.0 (CC BY 3.0) (http://creativecommons.org/licenses/by/3.0/)

Capítulo 3

Valores do eXtreme Programming

O eXtreme Programming define cinco valores para que seus papéis e práticas funcionem em sinergia de acordo com sua essência ágil: a comunicação, o feedback, a simplicidade, a coragem e o respeito; que serão descritos a seguir neste capítulo.

3.1 Comunicação

Ter foco na comunicação é essencial em projetos de software, pois é a principal forma de transmitir e trocar informações e conhecimentos. Por essa razão, é importante incentivar meios eficazes de comunicação, cujo valor está presente no Manifesto Ágil e na maioria das práticas de XP.

Ela pode ocorrer de várias formas, como, por exemplo: e-mail, telefone, skype (ou outras ferramentas de chat instantâneo), teleconferência, conversa face a face etc. Porém, é preciso entender que nem todas possuem a mesma efetividade.

A forma mais eficiente é a conversa face a face, usando um quadro branco como apoio para rabiscar ideias e/ou rascunhos sobre arquitetura de software. Distribuir quadros brancos no ambiente de desenvolvimento pode funcionar muito bem para estimular discussões de arquitetura, fluxos, algoritmos e, assim, aumentar a possibilidade de implementar certo logo na primeira tentativa.

A comunicação incentiva diretamente outro valor essencial no XP: o feedback.

3.2 Feedback

Na adoção das práticas, o feedback é realizado a todo momento, seja em relação aos requisitos do cliente, ao resultado da execução de testes unitários, ou na compilação do código na integração contínua. A compreensão das necessidades dos usuários e do negócio propriamente dito é um aprendizado constante. A razão de adotarmos estratégias iterativas e incrementais é que isso permite que os inevitáveis erros das pessoas sejam descobertos o mais cedo possível e reparados de forma metódica.

3.3 Simplicidade

A simplicidade é outro valor presente nas práticas XP, e a chave e a diretriz de um projeto ágil. Ela visa manter o trabalho o mais simples e focado possível, entregando somente o que realmente agrega valor. Acrescentar suporte para futuras funcionalidades de forma desnecessária complica o *design* e eleva o esforço para desenvolver incrementos subsequentes.

Infelizmente, projetar um design simples não é algo fácil, pois muitos times já estão acostumados com a prática de futurologia, sofrendo da Síndrome de Nostradamus; ou seja, mesmo sabendo o que o cliente quer hoje, eles insistem em tentar desenvolver uma solução que também resolva problemas

futuros. O ponto é: não temos certeza sobre o que vai acontecer no futuro, então precisamos nos focar *apenas* nos problemas e necessidades atuais do nosso cliente.

Lembre-se: busque sempre desenvolver o suficiente de forma simples e com qualidade.

3.4 Coragem

A coragem também é incentivada por meio do uso das práticas do XP. Por exemplo, o desenvolvedor apenas se sentirá confiante para refatorar o código criado por outro colega caso o time tenha um padrão de codificação e posse coletiva do código. Além dessa prática, o uso de controle de versão e testes unitários também encorajam isso. Além desses medos naturais de alteração de código, tem-se uma série de preocupações que exercem influência nos processos de desenvolvimento.

O cliente teme:

- Não obter o que foi solicitado/contratado;
- Pedir a coisa errada;
- Pagar demais e receber pouco;
- Jamais ver um plano relevante;
- Não saber o que está acontecendo (falta de feedback);
- Ater-se às primeiras decisões de projeto e não ser capaz de reagir à mudança do negócio.

Já o desenvolvedor teme:

- Ser solicitado a fazer mais do que sabe fazer;
- Ser ordenado a fazer coisas que não façam sentido;
- Ficar defasado tecnicamente;

- Receber responsabilidades, mas sem autoridade;
- Não receber definições claras sobre o que precisa ser feito;
- Sacrificar a qualidade em função do prazo;
- Resolver problemas complexos sem ajuda;
- Não ter tempo suficiente para fazer um bom trabalho.

Um verdadeiro time XP é composto de indivíduos corajosos que confiam em suas práticas, bem como nos seus colegas de time. A coragem faz-se ainda mais necessária nos momentos de crise.

3.5 Respeito

O respeito pelos colegas de time e pelo cliente é muito importante. Pessoas que são respeitadas sentem-se valorizadas. Os membros do time precisam respeitar o cliente; o cliente precisa respeitar os membros do time; e, além disso, o cliente deve fazer parte das decisões.

Os desenvolvedores nunca devem realizar mudanças que quebrem a compilação e que fazem os testes de unidade falharem ou realizarem outras ações que possam atrasar o trabalho ou o progresso do time.

Todos no time devem respeitar a posse coletiva do código, sempre primando pela qualidade e buscando um design simples por meio da refatoração.

A adoção dos demais quatro valores do XP conduzirá a este quinto valor: o respeito. Isso vai garantir um alto nível de motivação, além de incentivar a lealdade entre todas as pessoas envolvidas no projeto, criando um verdadeiro senso de trabalho em equipe.

O ponto é: como se adquirem confiança e respeito? Obviamente, entregando softwares que funcionam. E mais que isso, o respeito é fundamental para uma relação transparente e duradoura. É isso que realmente forma a parceria e a colaboração de todos envolvidos, algo essencial para a entrega contínua de valor.

Capítulo 4

Papéis do eXtreme Programming

O time XP é formado por papéis com objetivos diferentes, porém complementares, tais como: o papel de desenvolvedor (programador), do cliente, do gerente, do *coach*, do testador, do *tracker* e do *cleaner* [6].

Uma pessoa pode assumir mais de um papel; por exemplo, ser desenvolvedor e coach, ou, então, desenvolvedor e tracker.

Porém, é aconselhável que alguns papéis, como coach e gerente, não sejam assumidos por uma mesma pessoa, porque seus interesses são conflitantes. Ou seja, o gerente em determinados momentos pode sofrer uma pressão muito grande para entregar mais softwares e de forma mais rápida, e isso pode fazer com que, inconscientemente, ele repasse essa pressão para o time e acabe deixando de lado as práticas ágeis do XP; enquanto o coach deve justamente manter o time atuando de forma ágil e disciplinada, mantendo a agilidade mesmo em momentos de crise.

Nas próximas seções, falaremos mais sobre cada um dos papéis que compõem um time XP.

4.1 Desenvolvedor

O desenvolvedor, também denominado programador, é o coração do XP. O desenvolvedor multidisciplinado é um profissional capaz de trabalhar em todas as etapas do desenvolvimento de software, desde a escrita de histórias de usuário até o *deploy* em produção. No time XP, ele é um programador que estima as histórias de usuários e tarefas, quebra as histórias em tarefas, escreve testes, escreve código, automatiza processos de desenvolvimento e, gradualmente, aprimora o design do sistema.

Especificação, prototipação, design, desenvolvimento, testes e atividades ligadas ao DevOps são tarefas que o desenvolvedor poderá exercer em seu dia a dia. Desenvolvedores multidisciplinados elevam o nível de agilidade de um time, uma vez que todos podem vir a atuar em atividades que, eventualmente, tornam-se gargalos do projeto. Visto que eles escrevem código de produção em pares, precisam ter boas habilidades sociais e de relacionamento.

4.2 Cliente

Também conhecido como *o dono do ouro*, o cliente define e prioriza as histórias de usuário, validando o produto desenvolvido por meio de testes de aceitação. É importante que ele esteja o mais próximo possível do time, com disponibilidade suficiente para conversar e tirar as dúvidas habituais.

O nosso cliente nem sempre será o usuário; ou seja, ele pode conhecer o negócio, definir e priorizar os requisitos, mas não utilizará o software efetivamente. Quando esse cenário ocorrer, é necessário aproximar do time alguns desses usuários, pois, dessa forma, conseguimos priorizar os requisitos e validá-los com eles. São as pessoas que realmente utilizarão o software no dia a dia, afinal.

4.3 Coach

É o técnico do time XP. Ele orienta a todos, mantendo a disciplina na aplicação das práticas ágeis, e lembra a equipe da importância das cerimônias (como a reunião em pé, o jogo do planejamento e a reunião de retrospectiva, tudo que veremos mais à frente), da construção e manutenção de artefatos e do uso de ferramentas.

O coach pode facilitar reuniões e orientar o time em relação a que caminho seguir quando surgirem dúvidas ou impasses ligados à forma de trabalho.

É saudável que ele não seja a mesma pessoa que atua como gerente, porque os interesses desses papéis são diferentes. Enquanto o coach preocupa-se em manter o time engajado e disciplinado, o outro importa-se em priorizar e entregar o software para o cliente. Se uma mesma pessoa acumular os dois papéis, em momentos de crise, o gerente poderá falar mais alto e atropelar as práticas ágeis.

4.4 Testador

O testador no time XP auxilia o cliente a escolher e escrever testes de aceitação, para, então, automatizá-los. Ele serve também como um programador coach em técnicas de testes. Para o time de desenvolvimento, o testador não é responsável por pegar erros triviais, isto é papel dos próprios desenvolvedores. Ele faz parte do time XP, não devendo ser uma pessoa isolada e trabalhando em outro local. Ele pensa no teste e na qualidade do produto como um todo, considerando também os que rodarão na integração contínua, auxiliando em par os programadores a resolver problemas do sistema.

4.5 Cleaner

O cleaner é um membro do time que assume o papel de limpar o código; encorajar os membros a praticar pequenas refatorações; reduzir a complexidade e acoplamento do código; e a aumentar a coesão dos métodos, realizando sua extração e tornando o código cada vez mais enxuto [63]. Ele também se mantém o tempo todo atento aos impedimentos do time e às dívidas técnicas, ajudando, assim, a garantir a qualidade do código.

São pré-requisitos para um bom cleaner: a excelência técnica, o conhecimento do negócio e da arquitetura, bem como uma boa didática para explicar aos desenvolvedores o motivo de cada refatoração para elevar o nível técnico de todos os membros do time.

O cleaner incentiva o grupo a cuidar da saúde e do bem-estar do código. Um bom candidato pode ser o líder técnico ou o arquiteto do produto. Ele também pode conduzir *Coding Dojos* ou outras dinâmicas para elevar o conhecimento técnico da equipe.

4.6 Tracker

Responsável por coletar as métricas de projeto, o tracker é capaz de contar uma história da iteração do início ao fim, por meio dos apontamentos que realizou e das informações que foram coletadas. Como um tracker, você é a consciência do time e deverá coletar informações sem perturbar o processo de desenvolvimento mais do que o necessário.

Ao final de cada iteração, ele pode gerar métricas que mostram o desempenho do time. O coach pode trabalhá-las com o grupo, buscando manter aquilo que estiver satisfatório e modificar o que não estiver indo bem.

4.7 Gerente

O gerente facilita a comunicação dentro de um time XP e coordena a comunicação com clientes, fornecedores e com o resto da organização. Ele gerencia e acompanha o planejamento do projeto, pois o planejamento no XP é uma atividade, e não uma fase; e auxilia na priorização das histórias de usuário, assim como no agendamento de reuniões e de demonstrações com o cliente e os usuários. Também gera relatórios e gráficos de acompanhamento do projeto e administra a infraestrutura necessária ao time (máquinas, licenças, espaços físicos etc.).

O gerente não é o chefe do time de desenvolvimento, mas pode (e deve) ser um líder servo, o que significa que ele reconhece e auxilia nas suas necessidades reais.

4.8 Outros papéis

O XP considera que outros papéis relacionados ao negócio podem fazer parte do time, podendo incluir: usuários, consultores, chefes e executivos [7]. Porém, por tratarem-se de papéis complementares do XP, eles não serão abordados por nós.

Capítulo 5

Time coeso

"A aprendizagem não é obrigatória... Nem a sobrevivência."
– William Edwards Deming

Você se lembra do primeiro valor do Manifesto Ágil? *Indivíduos e interações mais que processos e ferramenta.* Você já parou para pensar por que esse é o primeiro valor?

Comumente nas organizações, diversos processos e ferramentas são desenvolvidos para tentar resolver problemas que, na realidade, são causados por lacunas na comunicação, na confiança das pessoas e na falta de motivação dos indivíduos, criando formalismos que engessam o trabalho e geram novos problemas. Esse primeiro valor enfatiza a importância das pessoas e do trabalho em time para um melhor desenvolvimento de software.

Eis um dos pontos-chaves no XP: em um projeto, todos fazem parte do time, integrando-se os clientes à equipe de desenvolvimento. Dessa forma, o

grupo coeso possuirá todas as competências de negócio e técnicas para desenvolver o software.

Um time efetivamente coeso maximiza os valores do XP: possui maior comunicação por estar integrado, busca continuamente a simplicidade e constrói confiança para ter-se coragem e gerar feedback por meio do respeito mútuo. A multidisciplinaridade, a auto-organização, a proximidade física e a melhoria contínua são características essenciais para um time efetivamente coeso.

5.1 Time multidisciplinar

O trabalho multidisciplinar tem raízes nas células de produção do Sistema Toyota de Produção [41]. O conceito de células de produção defende o trabalho em equipe para a produção de itens com características similares, sendo uma grande inovação dos japoneses na década de 70. Isso revolucionou a ideia taylorista-fordista de ter-se apenas um homem por estação de trabalho executando apenas um tipo de tarefa especializada.

O trabalho em célula de produção cria sinergia entre os indivíduos e otimiza o desempenho do processo, eliminando diversos desperdícios, tais como: espera; superprodução; estoque; defeitos de qualidade; movimentos; transportes; processos desnecessários; e, principalmente, o não uso da criatividade dos funcionários. É importante salientar que o membro de time XP pode assumir mais de um papel, o que foi abordado com mais detalhes no capítulo 4.

5.2 Time auto-organizável

Os times coesos são auto-organizáveis; ou seja, são responsáveis pelo seu próprio sucesso. Não há a necessidade de controle das atividades de cada indivíduo do time por um gerente de projetos ou um coordenador. A gestão é tão importante que não fica nas mãos de apenas uma pessoa. O gerente empodera o time nas suas decisões e trabalha no macrogerenciamento. O microgerenciamento está dentro do próprio time; não há ninguém melhor do que a própria equipe para tomar suas decisões de trabalho. O coach cuidará do método e

da liderança do grupo, porém a liderança situacional pode emergir de seus membros, dependendo da questão a ser resolvida.

5.3 Sentando lado a lado

O espaço físico influencia diretamente a comunicação e o trabalho do time coeso. É essencial que o time sente-se junto no mesmo ambiente físico. No início do dia, os membros realizam as reuniões diárias face a face e, durante ele, a programação em pares é feita constantemente. O ambiente de trabalho no XP contém quadros para a comunicação e feedback de informações importantes do projeto a todos envolvidos nele.

5.4 As reuniões de retrospectiva e a melhoria contínua

Em desenvolvimento de software, existe apenas o aperfeiçoar, e não a perfeição. Não há processo perfeito, nem projeto e nem histórias de usuário. Contudo, a melhoria contínua é buscada no dia a dia, sendo possível **aperfeiçoar** seu processo, seu projeto e suas histórias.

Um dos pontos-chaves de um time de alto desempenho está nas suas reuniões de retrospectiva, pois são identificados progressos a serem realizados no seu trabalho para aprimorar suas práticas. Essa reunião exige uma profunda reflexão do próprio trabalho para que se possam encontrar melhorias. No Lean, essa autorreflexão é chamada de *Hansei* e o ato de realizá-las é chamado de *Kaizen* [41].

Qual é a abordagem do XP para a melhoria contínua? É simples: conserte o XP quando ele falhar. O eXtreme Programming assume que não trará todas as respostas para o desenvolvimento de software, pois sabemos que não há bala de prata [28]. As regras devem ser seguidas até o time precisar mudá-las.

Os projetos são encorajados a iniciarem seguindo as regras do XP e, então, a adaptarem qualquer aspecto que não esteja funcionando, tudo isso em um ambiente de experimentação e avaliação. As reuniões frequentes de retrospectiva farão com que a melhoria contínua seja uma parte natural do desenvolvimento de software. O livro *Agile retrospectives* traz toda a base, além

de muitas dicas para conduzir retrospectivas [40].

O clima descontraído e divertido nas retrospectivas ajuda bastante os membros do time a expor problemas e tornar a melhoria contínua mais leve. As dinâmicas lúdicas com o uso de metáforas também ajudam muito na exposição de problemas e na coragem para dar passos em direção ao aperfeiçoamento, pois estimulam a criatividade. Paulo Caroli e Tainã Caetano trazem diversas dessas retrospectivas divertidas no livro *Fun retrospectives* [11].

O segredo das práticas do XP está na sinergia: uma prática sustenta e potencializa a outra. Por exemplo, as histórias de usuário permitem a comunicação para a criação de testes de aceitação e pequenas iterações, agilizando as entregas para obter-se feedback do cliente sobre o software sendo desenvolvido. Uma dica é utilizar a melhoria contínua também para a união das práticas.

Dica dos autores: qual o tamanho ideal de um time?

Reconhecemos que, mais importante do que o tamanho do time, é a sua formação: manter o foco por meio da atenção constante e buscar o comprometimento de todos. Porém, é quase unanimidade que grupos de sete pessoas variando mais ou menos duas normalmente funcionam bem, principalmente organizando e colocando as práticas em andamento. Isso facilita a realização das cerimônias e a dar os primeiros passos. Conhecemos times pequenos (4 a 6 pessoas) que não funcionavam muito bem, e também já trabalhamos com times maiores (13 ou 14 pessoas), nos quais o trabalho funcionava muito bem e de uma forma ágil e disciplinada. Na dúvida, experimente!

Capítulo 6

Cliente presente

"Clientes não esperam que você seja perfeito. Eles esperam que você resolva coisas que eles fizeram errado."
– Donald Porter, vice-presidente da British Airways

Uma das necessidades básicas do XP é ter o cliente presente, pois isso faz com que ele se sinta parte do time, o que agiliza o trabalho dos programadores. Desse modo, esse envolvimento é *extremo* com a comunicação e com o feedback.

Todas as fases do XP precisam da comunicação com o cliente, de preferência face a face. Por isso, é interessante que ele esteja no próprio local do desenvolvimento do software. Escrever especificações leva um tempo muito longo e elas não comunicam efetivamente bem, tanto que poucas pessoas realmente as leem por completo. É muito mais efetivo conversar sobre o que será necessário no software para seu desenvolvimento já se iniciar.

O cliente tem uma participação essencial no *jogo do planejamento*, escrevendo e priorizando histórias e discutindo detalhes dos requisitos diretamente com o time para criarem as tarefas de implementação (vide capítulo 10). Se houver itens importantes a serem documentados, registre-os como critérios de aceitação; assim, eles poderão ser úteis por serem passíveis de automatização. Ele também precisa estar disponível também para discutir os testes de aceitação para que o time valide as histórias.

Uma objeção comum sobre ter-se o cliente presente é que ele requisitará apenas o que é de seu interesse e não pensará no negócio como um todo. Isso é de responsabilidade do trabalho dele e de outras pessoas ligadas ao negócio, e poderá acontecer de qualquer forma, com ou sem sua presença.

Ter somente um representante do negócio (ou até não possuir cliente) pode ser um problema, pois faz com que funcionalidades sem utilidade sejam desenvolvidas e que critérios de aceitação não realistas sejam criados. O time de desenvolvimento trabalhará por itens sem valor real que não fazem parte da solução real.

Tudo é questão de prioridade

É importante entendermos que desenvolver software é, acima de tudo, aprendizado de todos. Para isso, é necessária muita colaboração. Ninguém desenvolve sozinho. Há alguns anos, estávamos começando um projeto com um novo cliente. Como de praxe, explicamos como funcionava o nosso trabalho e como a participação dele era importante neste processo. Era uma forma de sensibilizar e mostrar o quão importante estar ativo no desenvolvimento.

Quando começamos o trabalho, tudo aquilo que havíamos definido foi perdendo força e ele distanciou-se. Chegamos a um ponto em que tivemos que questioná-lo sobre sua participação (ou de alguém que pudesse ajudar). Ele simplesmente nos disse que não tinha tempo e que estava resolvendo outros problemas na empresa. Ficou claro que aquele era o momento de parar. E foi o que fizemos. Simplesmente, parou-se o desenvolvimento. Dissemos-lhe que, se ele não tinha tempo para envolver-se (ou envolver alguém que pudesse representá-lo), este projeto não era tão importante assim para ele e, portanto, ele não deveria gastar tempo e, principalmente, nem dinheiro com isso. Simples assim.

6.1 O que fazer quando o cliente não pode estar presente?

Pessoas de negócio podem ser bem requisitadas em seu trabalho e não terão tempo disponível para estarem presentes no local do desenvolvimento. Espere, desde o início, lacunas no entendimento de ambas as partes, tanto sobre o negócio quanto sobre o desenvolvimento. Nesses casos, é necessário maximizar a comunicação buscando outras alternativas de tecnologia e de processo de desenvolvimento.

Utilize ao máximo a comunicação por telefone e reuniões virtuais. O trabalho por reuniões remotas é uma boa alternativa. Ajudará também ter um representante que entenda do negócio e tenha tempo e acesso ao cliente. Tente ao menos que ele compareça nas reuniões de planejamento. E mais, se for

possível, aproveite ao máximo seu tempo e faça um planejamento presencial intensivo de um dia inteiro ou de uma semana inteira.

Dica

Faça frequentemente pequenas entregas. Essa é uma boa estratégia para alinhar o software com o negócio. Se o cliente não pode ver o time, que ao menos veja o sistema. ;)

Capítulo 7

Histórias de usuário

"Histórias não são requisitos; elas são discussões sobre resolver problemas para nossa organização, nossos clientes e nossos usuários que levam a acordos sobre o que construir."
– Jeff Patton

Histórias de usuário (*user stories*) descrevem seus requisitos em uma forma ágil. Para discutir seus detalhes, é necessária a comunicação face a face entre o time e o cliente, encorajando o trabalho em conjunto. Cada história segue um ciclo de vida, normalmente nascendo como um épico (histórias grandes), sendo detalhada de acordo com sua prioridade. Elas são textuais, não necessitam de ferramenta específica, são compreensíveis por todos do desenvolvimento ou do negócio, e são descritas em cartões, também chamadas de cartões de história (*story card*). Outras formas de documentação complementares podem ser utilizadas juntamente, o XP não impede isso desde que

sejam úteis e necessárias, tais como protótipos ou diagramas.

Veja alguns exemplos de histórias de usuário:

- Como um contador, eu quero registrar uma movimentação de patrimônio;
- Como uma leitora, quero pesquisar livros por categoria;
- Como repositor de estoque, quero localizar quais as prateleiras do supermercado que contêm menos de 50% de sua capacidade de produtos.
- Como cliente, quero encontrar as pizzarias na minha cidade que estejam abertas na segunda-feira à noite.

A essência da utilização de histórias de usuários como requisitos está de acordo com os valores do Manifesto Ágil.

- **Indivíduos e interações mais que processos e ferramentas:** elas enfatizam a conversação, não dependendo de processo ou de ferramenta;
- **Software em funcionamento mais que documentação abrangente:** elas vão direto ao ponto com uma forma flexível de documentação para o que há de maior valor para o negócio;
- **Colaboração com o cliente mais que negociação de contratos:** por serem sucintas e necessitarem da conversação do cliente com os desenvolvedores, permitem a negociação e colaboração com o negócio;
- **Responder a mudanças mais que seguir um plano:** elas respondem às mudanças por serem flexíveis em seu ciclo de vida, dando base para as alterações quando necessárias.

Nas seções a seguir, detalharemos sobre o modelo 3C e o ciclo de vida de histórias. Também mostraremos como escrever usando um modelo de escrita, refinando com INVEST, quebrando as histórias em tarefas SMART e utilizando personas; diversas dicas de escrita de boas histórias; e um catálogo de *bad smells* de histórias.

7.1 Modelo 3C

O tamanho de um cartão é suficiente para detalhar um requisito de usuário? Sim, pelo fato de as histórias serem utilizadas como lembretes para a conversação do time e para a confirmação do cliente. Os cartões de histórias de usuário são como slides: da mesma forma que um instrutor ensina por meio deles, um cliente mostra os requisitos do sistema por meio de histórias de usuário.

Para deixar claro o propósito das histórias de usuário, Ron Jefries criou o modelo 3C [31]. Cada uma das três letras *C* representa um aspecto crítico delas:

- ***Cartão:*** as histórias de usuário são escritas em cartões (ou no tamanho de um cartão). Ele não conterá toda a informação de um requisito, mas serve para lembrar a todos do que este se trata. As prioridades e os custos também podem ser anotados neles.

- ***Conversação:*** o requisito é comunicado do cliente ao time por conversação (face a face), podendo ser suplementado por documentos.

- ***Confirmação:*** a confirmação da história dá-se por exemplos para criar testes de aceitação; ou seja, pelos critérios de aceitação. O cliente dirá ao time como aceitará cada história por meio de critérios de aceitação. Essa é uma ótima forma de suplementar a documentação. A preferência é de que esses critérios sejam automatizados pelo time. A história de usuário estará pronta quando todos eles estiverem passando nos testes de aceitação.

7.2 Ciclo de vida

Cada história segue um ciclo de vida, sendo refinada progressivamente. Elas normalmente nascem em épicos e são agrupadas em temas. Um épico pode ser visto como uma história muito grande [49] que pode levar diversas iterações para serem desenvolvidas e que ajudam a esboçar a funcionalidade do produto sem se comprometer com detalhes. Assim que priorizado, ele é quebrado em diversas histórias.

Um tema representa uma capacidade do produto ou um objetivo, ajudando a mostrar sua completude e também a encontrar as histórias certas. Uma estratégia é priorizar os temas para, então, priorizar as histórias relacionadas. Aquelas de maior prioridade terão seus critérios de aceitação completos e, depois, serão refinadas com INVEST (explicado a seguir neste capítulo) e quebradas em tarefas de implementação.

Um exemplo de épico pode ser: *como um repositor de estoque, eu quero gerenciar a reposição do supermercado*, fazendo parte do tema "Gestão de estoque". Uma história que pode nascer a partir dele é: *como repositor de estoque, quero localizar quais as prateleiras do supermercado que contêm menos de 50% de sua capacidade de produtos.*

7.3 Utilizando um modelo de escrita

Ter um modelo é útil para a escrita de histórias. Existem alguns tipos de modelos e variações; um dos mais utilizados é o da Connextra (vide imagem 7.1), que traz as informações principais no cartão:

- **Como um (*As a...*)**: para quem será útil a história, para qual usuário protagonista do sistema; pode também ser uma persona (veremos em 7.6);
- **Eu quero (*I want...*)**: o que a história está especificando de requisito, a necessidade do usuário propriamente dita;
- **Para que (*So that...*)**: razão de negócio, o motivo pelo qual a história existe; mostra o valor de negócio da história.

Fig. 7.1: Modelo original de cartão de história de usuário da Connextra. Fonte da imagem: http://agilecoach.typepad.com

Basicamente, Connextra gera um modelo de escrita, ilustrado a seguir.

<Título>

Como um <papel de usuário>

Eu quero <objetivo>

Para que <razão de negócio>

Fig. 7.2: Modelo de escrita de história de usuário

Como exemplo, utilizando-se esse modelo, a história *Como repositor de estoque* ficará dessa forma:

Reposição de estoque

Como um repositor de estoque
Eu quero localizar quais prateleiras do supermercado que contêm menos de 50% de sua capacidade de produtos
Para que eu possa fazer a reposição dos produtos e não faltar itens para os clientes em compras.
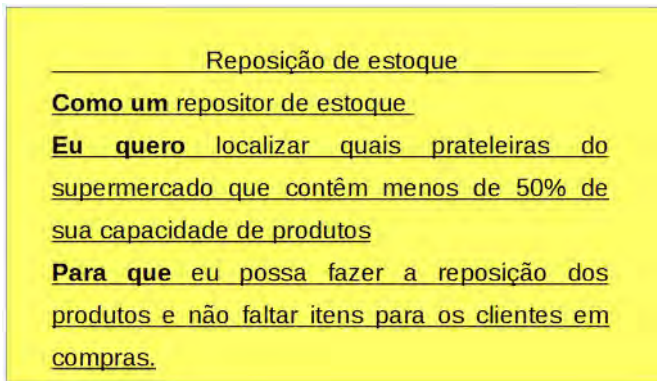
Fig. 7.3: Exemplo de uma história de usuário

7.4 Refinando com INVEST

Muitas vezes, as histórias de usuário precisam ser lapidadas, pois podem estar complicadas de entender, grandes demais para serem desenvolvidas, dependentes demais para serem entregues, ou podem não ser verificadas. Para esse refinamento, utiliza-se o acrônimo INVEST, o qual traz 7 características. Aplicar todas elas não é tão fácil; até dizem que INVEST é o nirvana das histórias de usuários, mas deve-se tentar ao máximo obtê-las.

Refinando com INVEST, uma história de usuário deve ser:

- **I**ndependente de outras histórias para facilitar a negociação, a priorização e a implementação;
- **N**egociável com o cliente, para que o time possa manter um ritmo sustentável para a entrega contínua de valor;
- **V**alorosa, assim cada entrega agregará grande valor ao negócio do cliente;
- **E**stimável, desse modo o time poderá estabelecer quais histórias serão implementadas de acordo com sua velocidade;
- ***S**mall* (pequena) o suficiente para ser implementada dentro de uma iteração, sem correr grandes riscos de não a completar;

- **T**estável, apenas assim poderá ter qualidade na entrega e certeza de que o cliente aceitará a história como pronta.

7.5 Implementando com tarefas SMART

Cada história de usuário possuirá tarefas a serem desenvolvidas, como: implementação, teste, integração e deploy. Quebrá-las em tarefas gera mais certeza do trabalho que o time precisará entregar. Cada uma delas deverá possuir as 5 características do acrônimo SMART:

- e**S**pecífica o suficiente para ser entendida, para que se implemente exatamente o que é necessário, evitando a sobreposição de outras tarefas;
- **M**ensurável, possuindo um tamanho para garantir que a tarefa seja concluída (contabilizando testes e refatoração);
- **A**lcançável para que todos possam ter acesso às informações necessárias para sua conclusão;
- **R**elevante, contribuindo para a implementação da história;
- ***T**ime-boxed*, limitada a uma duração específica (não sendo uma estimativa formal, mas uma expectativa de desenvolvimento, inclusive para pedir ajuda). Tarefas com grande duração devem ser evitadas, podendo serem divididas em tarefas menores ou sendo revisadas.

7.6 Personas

Uma forma recomendada para pensar mais no usuário do sistema é utilizar o conceito de personas, que é a caracterização de um papel de usuário do sistema. Uma persona descreve as qualidades, os valores, o comportamento e os objetivos de usuários alvos no sistema. Uma dica é dar nomes às personas que lembrem seus papéis; por exemplo, o nome **Ademar** para a persona relativa ao papel de administrador. Na escrita da história, escreve-se o nome da persona, em vez do papel de usuário.

7.7 Story points

Story points é o nome comum para o tamanho de histórias de usuário. A velocidade do time é dada contabilizando a média de todos os *story points* realizados nas iterações. Dentre as diversas variações de pontuar uma história, a forma mais efetiva é dimensioná-las relacionando com as outras, utilizando a experiência no passado para determinar o quanto pode ser feito em uma iteração [27]. O time pode utilizar uma sequência não linear de valores, como a de múltiplos de dois (1, 2, 4, 8, 16, ...) ou a de Fibonacci (1, 2, 3, 5, 8, 13, ...).

O que realmente interessa é que se possa prever quantas histórias poderão ser entregues nas iterações, diminuindo o risco de termos problemas no desenvolvimento. Há os que acham melhor não estimar, mas apenas contabilizar, em média, quantas serão realizadas a cada iteração [36]. Existem times que até utilizam tamanhos de camiseta (P, M, G, GG), e outros que usam dias ideais ou horas de desenvolvimento [13].

Por estar confuso com *story points*, um cliente de Joshua Kerievsky criou o termo NUTs (*Nebulous Units of Time* Unidades Nebulosas de Tempo) [36]. Independente da forma de encontrar o tamanho da história, é o grupo que decide suas estimativas e sua velocidade.

7.8 Escrevendo boas histórias

A prática com boas orientações é importante para escrever uma história útil para o time e para o negócio. Mike Cohn, em seu livro *User stories applied*, traz orientações para escrever boas histórias de usuário [13]. Complementamos com mais algumas dicas:

- **Comece com as histórias objetivas:** os usuários do sistema possuem objetivos diretos no sistema; comece escrevendo as histórias que resolvam os problemas desses objetivos.

- **Fatie o bolo:** escreva histórias que possuam uma camada de cada fase de implementação de uma funcionalidade, desse modo entregará valor. Imagine uma história como uma fatia de bolo, em que cada camada é a parte de projeto, codificação, teste, integração e deploy.

- **Escreva histórias fechadas:** faça com que a história entregue valor real ao negócio, algo que conclua o processo do usuário.
- **Coloque as restrições nos cartões:** as restrições são os requisitos não funcionais, como de desempenho, de carga, de segurança e de usabilidade. Elas podem ser descritas nos cartões na forma de critérios de aceitação.
- **Escreva suas histórias no horizonte:** utilize o ciclo de vida das histórias de usuário. Faça com que elas surjam de épicos. Conforme a prioridade, os épicos são quebrados em histórias e, então, é iniciada a especificação dos critérios de aceitação relacionados.
- **Evite a interface de usuário o maior tempo possível:** a essência dos detalhes da interface de usuário é de projeto, e não de requisito de usuário, na maioria das vezes. Detalhá-la torna-se necessário conforme a prioridade aumenta.
- **Algumas coisas não são histórias:** histórias são para escrever requisitos de usuário, e não para documentar a ajuda do sistema, por exemplo.
- **Inclua os papéis de usuários:** a utilização de um modelo ajuda nisso, pois ele lembra de escrever para quem a história servirá.
- **Escreva para a persona protagonista:** muitas vezes, uma história de usuário servirá para mais de uma persona ou mais de um papel de usuário. Por essa razão, escreva a história para a persona ou papel de usuário que mais agregará valor, de maior propósito.
- **Escreva em voz ativa:** escrever em voz passiva pode tornar a história confusa.
- **O cliente escreve:** apenas dessa forma não haverá lacunas na escrita. Quando isso não for possível, faça com que ele leia, valide e discuta todas as histórias.
- **Dê títulos às histórias:** um título ajudará na sua identificação. Não enumere apenas os cartões de história, pois é muito estranho conversar

sobre a "história 32" (como se fazia antigamente com nomes de tabelas de banco de dados).

- **Não esqueça do propósito:** lembre-se sempre do propósito de uma história de usuário, que é servir como um lembrete a todos (time e cliente) para a comunicação do requisito.
- **Descreva os defeitos (*bugs*) como critérios de aceitação:** quando houver um *bug*, provavelmente é porque faltou ter escrito (ou automatizado) um critério de aceitação. Escreva um critério de aceitação que falhe diretamente no problema.

7.9 Catálogo de story smells

Para escrever boas histórias, também é necessário identificar e evitar *bad smells* (maus cheiros). O livro *User stories applied* do Mike Cohn possui um catálogo desses *smells* [13], que ocorrem normalmente quando:

- **A história depende de várias outras histórias:** a interdependência impossibilita a negociação e a priorização das histórias;
- **A história está muito curta:** histórias são sucintas, mas não é para serem muito curtas;
- **A história possui muitos detalhes:** repetindo que histórias são sucintas. Os detalhes normalmente serão discutidos entre o time e o cliente, lembre-se do modelo 3C;
- **Há funcionalidades não necessárias:** lembre-se do valor do XP: simplicidade. Quando houver funcionalidades desnecessárias em histórias, é um grande indicador de que não se está pensando em simplicidade;
- **O cliente não escreve, não confirma e não prioriza as histórias:** esse é um dos mais comuns e piores maus cheiros;
- **Não está explícito o valor do negócio:** escrever a razão de negócio ajuda a demonstrar seu valor. Um modelo ajuda nisso. Histórias com

pouco valor não podem ser priorizadas. Provavelmente, o problema está em uma lacuna na realização do jogo do planejamento;

- **É pensado muito à frente nas histórias:** isso atrapalha a agilidade, pois mudanças poderão vir logo, então não se deve detalhar muito à frente. Deve-se fazer somente o necessário para as histórias priorizadas;
- **É detalhada a interface de usuário muito antecipadamente:** deve-se pensar na interface de usuário quando a história for priorizada. Utilize o tempo presente para trabalhar com as histórias já priorizadas;
- **Existir detalhes específicos de tecnologia, projeto de banco de dados e algoritmos:** detalhes de projeto e de implementação não fazem parte de requisitos. Esses detalhes deveriam ser discutidos entre o time, ou então em outros formatos de especificação, tais como diagramas e protótipos.

Capítulo 8

Testes de aceitação

"Testes de aceitação são mapas da estrada para a iteração, dizendo ao time aonde é preciso ir e quais pontos de referência olhar."
– Lisa Crispin

O propósito dos testes de aceitação é a comunicação, a transparência e a precisão [43]. Quando os desenvolvedores, os testadores e o cliente concordarem com eles, todos entenderão qual é o plano para o comportamento do sistema. Chegar a esse ponto é responsabilidade de todas as partes. Eles validam como o cliente aceitará as funcionalidades prontas, pois são testes funcionais que guiam o time no desenvolvimento para, então, colocar em produção o que foi decidido que o sistema deve conter.

Esses testes são criados a partir das histórias de usuários selecionadas pelas pessoas de negócio no planejamento de iterações (veremos no capítulo 10). A história de usuário apenas estará completa quando todos seus critérios

de aceitação estiverem passando. Auxiliado pelo testador do time, o cliente é quem especifica os exemplos dos critérios de aceitação na forma de cenários, que podem conter dados e variações que criticam a história [14]. Os exemplos de cenários esclarecem as histórias ao time.

8.1 Automatização

Testes de aceitação sempre têm de ser automatizados por uma razão simples: custo [43]. Caso você ache custoso ter testes de aceitação automatizados, aguarde para ver o custo da confusão e do *debug* de código ao longo prazo.

Para fazer pequenas entregas frequentes, é necessário que os testes de aceitação estejam automatizados e que rodem no *build* gerado pelo servidor de integração contínua. O cliente poderá decidir se uma história desenvolvida poderá ir para produção mesmo quando alguns testes de aceitação não estiverem passando. Apenas ele tem esse poder.

8.2 Validando com critérios de aceitação

Cada história de usuário possui um ou mais critério de aceitação, criado também pelo cliente. O testador ajuda-o a pensar em cenários bons de teste, sendo escritos já prevendo a automatização. Esses critérios descrevem como a história será aceita pelo cliente e servem de guia para o desenvolvimento.

Como exemplo, considerando a história *Como um repositor de estoque, quero localizar quais as prateleiras do supermercado que contêm menos de 50% de sua capacidade de produtos*, podemos escrever os critérios de aceitação:

- Deve exibir uma lista de prateleiras com o setor e a seção de cada uma, e o tipo e o nome do produto contido;
- Deve exibir as prateleiras com menos de 50% da capacidade de itens;
- Não deve exibir as prateleiras com 50% ou mais da capacidade de itens.

Os critérios de aceitação também podem seguir um modelo. Comumente, o utilizado é o de cenários de BDD (*Behaviour-Driven Development* Desenvolvimento Guiado por Comportamento) [47]. Esse modelo possui três informações:

- **Dado que:** precondição, são os passos para preparar para ação a ser validada;
- **Quando:** ação que vai disparar o resultado a ser validado;
- **Então:** resultado a ser validado.

Você encontrará bons exemplos de histórias com critérios de aceitação no livro *Histórias de usuário* [61]. Como um exemplo, o critério de aceitação *Deve exibir o setor e a seção da prateleira com o tipo e o nome do produto* usando o modelo, ficará desta forma:

- **Dado que** o usuário logado é um repositor de estoque;
- **Quando** o item de menu "Consultar prateleiras para reposição" for clicado;
- **Então** a lista de todas as prateleiras para reposição é exibida.

Capítulo 9

Liberação frequente de pequenas entregas

"Software funcionando é a medida primária de progresso."
– 7º princípio do Manifesto Ágil

Por que deixar o cliente esperando um longo tempo para entregar um requisito importante? No XP, cada *release* deve ser tão pequena quanto possível e com o maior valor de negócio. As pequenas entregas contêm as histórias de usuário identificadas e priorizadas no jogo do planejamento, que veremos a seguir. As histórias de usuários devem ser suficientemente granulares para caberem em uma entrega pequena. Manter a qualidade no desenvolvimento é essencial para poder entregar frequentemente. Em alguns casos, fica difícil liberar releases pequenas, precisando ser uma entrega do tipo *tudo ou nada*.

Diversos benefícios surgem quando se liberam frequentemente pequenas entregas com alto valor:

- Entrega de valor adiantado e contínuo;
- O processo é aprimorado rapidamente por falhar mais cedo (conceito *fail fast*);
- Feedback do cliente mais cedo para confirmação ou adaptação dos requisitos;
- Satisfação dos usuários por ter respostas rápidas às suas necessidades;
- Maior qualidade, confiança e estabilidade;
- Reduz a taxa de defeitos por precisar realizar testes completos em ciclos menores;
- Realiza um design simplificado e suficiente apenas para a entrega em questão;
- O código é atualizado e integrado com maior frequência;
- O software não fica ultrapassado;
- Maior engajamento do time por ver seu trabalho sendo útil e empoderado;
- Facilita enxergar os diversos desperdícios ocultados nas grandes entregas;
- Evita a procrastinação de prazos.

9.1 Foco na qualidade é o ponto-chave

A qualidade é o ponto-chave para entregar com frequência. Os testes automatizados e a integração contínua são essenciais para agilizar os builds das entregas com alta qualidade no software. Pequenas entregas maximizam os

benefícios da abordagem iterativa e incremental, evoluindo o software em pequenos pedaços. Times extremos conseguem desenvolver uma entrega a cada iteração ou a cada dia, tal como no conceito de entrega contínua [20].

Quando isso não for possível, as iterações auxiliarão no acompanhamento do andamento do desenvolvimento da release. De qualquer modo, se não houver um ótimo nível de qualidade em cada build do software, mais e mais defeitos voltarão e menos se poderá entregar nas próximas releases. O foco na qualidade é essencial para entregas pequenas e frequentes.

9.2 Releases tudo ou nada

Ok, sabemos que alguns sistemas são *tudo ou nada*; ou seja, para colocar em uso efetivo é necessário que se faça uma grande entrega. Entretanto, até para esses sistemas é valioso pensar se há algum modo de quebrá-los em entregas menores. As solicitações de mudança virão naturalmente quando o usuário utilizar o software, então é melhor ter um processo já orientando a mudanças. Entregas grandes não combinam com um planejamento ágil orientado a mudanças. Lembre-se do quarto valor do Manifesto Ágil: *responder a mudanças mais que seguir um plano*.

Quanto mais tempo aguardamos para liberar uma funcionalidade aos usuários, menos tempo você terá e mais complexo será para adaptá-la ou corrigi-la. Na visão enxuta [41], a liberação frequente de pequenas entregas diminui consideravelmente os desperdícios de estoque, defeitos e tempo de espera (do cliente). Isso gera fluidez e reduz o tempo de ciclo das entregas (tempo entre o fim ou o início de produção de dois itens). Continuamente, persista na tentativa de diminuir o tamanho de suas entregas.

Capítulo 10

O jogo do planejamento

"Planos não são nada. Planejamento é tudo."
– Dwight David "Ike" Eisenhower

Você já trabalhou em um projeto *marcha da morte* [65] que é destinado a falhar e que você não podia nem sequer dar opiniões técnicas? Um projeto com tantos problemas que parecia uma bola de neve despencando montanha abaixo? No XP, isso não acontece.

O jogo do planejamento envolve os clientes e os desenvolvedores para planejarem as entregas de uma forma colaborativa. Também chamado de *release planning*, o nome *jogo* mostra a essência da dinâmica: tratar o planejamento como um jogo que contém um objetivo, jogadores, peças e regras. Seu resultado é maximizado por meio da colaboração de todos os jogadores. Ele agrega todo o processo necessário para a entrega de software como a escrita de histórias de usuário, a priorização pelo cliente e a criação das estimativas.

Visto que o trabalho de planejamento é encarado como um jogo, então quem seria o adversário? Talvez você pense que é o cliente. Errado! Ele é o maior interessado no sucesso do projeto. Afinal, já investiu tempo e dinheiro. O adversário está justamente no desafio em se conseguir entregar o melhor produto, dados as variáveis de projeto (prazos, custos, *time-to-market* etc.).

As pessoas de negócio definem o escopo, a prioridade, a composição e as datas das entregas. As decisões de negócio têm suporte das pessoas de desenvolvimento. Estas tomam decisões técnicas por meio de estimativas e de análises de consequências técnicas no negócio, e decidem seu processo de trabalho. Tanto o Negócio quanto o Desenvolvimento têm voz nas decisões das entregas. Não há bola de neve no XP. :)

O desenvolvimento iterativo e incremental é utilizado no jogo do planejamento. Os incrementos são definidos no planejamento de releases; e as iterações, no planejamento de iterações. Cada incremento representa uma entrega de valor em produção ao cliente, sendo desenvolvido em uma ou mais iterações. A divisão em iterações auxilia o time a quebrar as entregas em pedaços menores para planejamento e feedback de um modo simples e ágil.

Nas próximas seções, definiremos o jogo como um todo, com suas regras e comprometimentos. Falaremos também sobre como manter o foco e a aprendizagem no planejamento.

10.1 Definindo o jogo e suas regras

Os jogos e as regras do jogo do planejamento são:

- **Objetivo:** maximizar o valor do software produzido pelo time; ou seja, colocar em produção a maior quantidade de histórias de usuário com maior valor ao longo do projeto.

- **Estratégia:** o time deve investir o menor esforço para colocar a funcionalidade de maior valor em produção tão rápido quanto possível, em conjunto com estratégias de projeto e programação para reduzir o risco. Dividir para conquistar é a estratégia utilizada ao dividir o software em entregas frequentes, e cada entrega em iterações pequenas.

- **As peças:** as peças básicas do jogo são histórias de usuário e as tarefas de implementação. As histórias são as peças no planejamento das releases; já as tarefas, no planejamento das iterações.
- **Os jogadores:** as pessoas do Desenvolvimento e as pessoas de Negócio.
- **Os movimentos:** as regras dos movimentos existem para lembrar a todos de como eles gostariam de agir em um melhor ambiente com confiança mútua e dar uma referência para quando as coisas não estiverem indo bem.

10.2 Entendendo regras e comprometimentos

Os movimentos do jogo são dados por suas regras e são divididos em duas partes: a primeira trata dos movimentos para as entregas (planejamento de releases); a outra dos movimentos para as iterações (planejamento de iterações), conforme apresentado na figura a seguir.
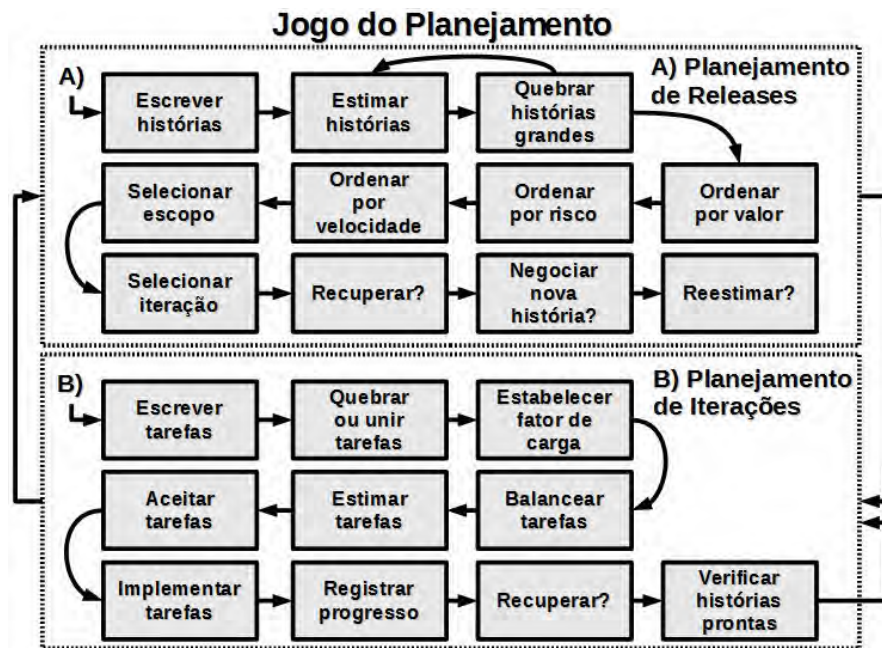
Fig. 10.1: Movimentos do jogo do planejamento divididos entre o planejamento de releases e de iterações

Planejamento de releases

Para uma release, o jogo do planejamento define as regras para o cliente direcionar o desenvolvimento na frequência de uma a três semanas. Seus movimentos para uma release são dados em três fases: **Exploração**, **Comprometimento** e **Direcionamento**.

Exploração: ambos os jogadores descobrem novos itens que o sistema pode fazer, dividindo-se em três movimentos:

- *Escrever uma história de usuário:* o Negócio escreve algo que o sistema deverá fazer;
- *Estimar uma história de usuário:* o Desenvolvimento estima o tempo ideal de programação para implementar a história;

- *Quebrar uma história de usuário:* caso o Desenvolvimento não consiga estimar a história ou sua estimativa for muito grande, o Negócio identifica sua parte mais importante para que o Desenvolvimento possa estimá-la e desenvolvê-la. Spikes podem ser utilizados para melhorar a estimativa da história (vide capítulo 11).

Comprometimento: o Negócio decide o escopo e o tempo para a próxima entrega de acordo com as estimativas, e o Desenvolvimento se compromete com a entrega, dividindo-se em quatro movimentos:

- *Ordenar por valor:* o Negócio separa as histórias em três categorias: as essenciais; as menos essenciais, mas de alto valor; e as que seriam agradáveis de se ter;
- *Ordenar por risco:* o Desenvolvimento separa as histórias em três categorias: aquelas que podem ser estimadas com precisão, aquelas que podem ser estimadas razoavelmente bem, e aquelas que não podem ser estimadas de qualquer modo;
- *Ordenar por velocidade:* o Desenvolvimento mostra ao Negócio quanto de tempo ideal de programação o time poderá trabalhar em um mês do calendário;
- *Selecionar escopo:* o Negócio escolhe as histórias para a entrega (trabalhando por tempo) ou uma data para as histórias a serem desenvolvidas (trabalhando por escopo), utilizando as estimativas realizadas pelo Desenvolvimento.

Direcionamento: o plano é atualizado no que foi aprendido pelo Negócio e pelo Desenvolvimento, dividindo-se em quatro movimentos:

- *Iteração:* a cada iteração (uma a três semanas), o Negócio seleciona uma com as histórias de maior valor a serem implementadas;
- *Recuperação:* caso o Desenvolvimento descubra que sua velocidade foi superestimada, ele pode pedir ao Negócio para encaixar apenas as histórias de maior valor de acordo com a nova velocidade;

- *Nova história:* caso o Negócio descubra uma nova história a ser adicionada à entrega que já está sendo desenvolvida, ele pode escrevê-la, o Desenvolvimento estimá-la, e, então, trocá-la por outra selecionada pelo Negócio;
- *Reestimar:* caso o Desenvolvimento descubra que o plano não é exato, ele pode reestimar todas as histórias restantes.

Planejamento de iterações

As peças são as tarefas de implementação e os jogadores são os programadores. As decisões sobre as iterações são mais flexíveis em questão de escopo e de prazos, pois dependem mais do Desenvolvimento; enquanto as sobre entregas dependem mais do Negócio. Suas fases são semelhantes às de uma entrega: **Exploração**, **Comprometimento** e **Direcionamento**.

Exploração: o Desenvolvimento cria suas tarefas e identifica o tempo de trabalho, dividindo-se em três movimentos:

- *Escrever uma tarefa:* transformar as histórias da iteração atual em tarefas;
- *Quebrar ou unir tarefas:* caso a tarefa esteja grande demais, quebre-a em tarefas menores. Caso ela esteja pequena demais, una-a a outra relacionada;
- *Estabeleça um fator de carga:* cada programador escolherá seu fator de carga para a iteração, o qual é a porcentagem de tempo que ele realmente estará desenvolvendo; ou seja, é removido o tempo em que se realiza outras, tais como reuniões do time.

Comprometimento: o Desenvolvimento compromete-se com suas tarefas, dividindo-se em três movimentos:

- *Aceitar uma tarefa:* o programador puxa a tarefa para si e aceita a responsabilidade por ela;
- *Estimar a tarefa:* o programador responsável pela tarefa a estima em Dias Ideais de programação;

- *Balanceamento:* cada programador soma suas tarefas e multiplica pelo fator de carga, balanceando sua carga de trabalho, caso esteja maior ou menor do que a estabelecida.

Direcionamento: o Desenvolvimento desenvolve as tarefas para verificar a história, dividindo-se em quatro movimentos:

- *Implementar uma tarefa:* o programador pega o cartão da tarefa, encontra um par para programar, desenvolve a tarefa com TDD (Desenvolvimento Guiado por Testes) e ao final integra o código fazendo passar a suíte de teste relacionada;
- *Registrar progresso:* a cada dois ou três dias, um membro do time atualiza o andamento das tarefas de cada membro, com o tempo gasto e o tempo que falta para cada tarefa;
- *Recuperação:* programadores que ficaram sobrecarregados pedem ajuda reduzindo o escopo das tarefas, reduzindo o escopo das histórias, removendo aquelas não essenciais ou pedindo ao cliente para postergar a história para outra iteração;
- *Verificar a história:* caso estiverem prontos, rodar os testes funcionais para as tarefas que já foram concluídas, complementando a suíte de testes funcionais com os novos casos de testes;

As histórias de usuários são estimadas antes de serem desenvolvidas. Já na iteração, o programador pega a tarefa para si e depois a estima. Nem todas estarão ligadas diretamente com o Negócio, mas sim com as necessidades do Desenvolvimento; por exemplo, para configurar um servidor de integração.

10.3 Mantendo o foco

Você lembra-se do objetivo do jogo do planejamento? Maximizar o valor do software construído pelo time para o cliente é o foco do jogo. É importante que todos participem (Desenvolvimento e Negócio) com colaboração. A simplicidade (a arte de maximizar a quantidade de trabalho não realizado)

é essencial para o feedback contínuo, sendo importante também para o time sentir-se valorizado por fazer entregas frequentes de valor. O Desenvolvimento e o Negócio devem manter um ritmo sustentável e constante de trabalho para manter o foco.

10.4 Todo momento é um momento de aprendizado

No jogo do planejamento, até quando se erra, aprende-se. O erro não é visto como algo ruim, desde que se aprenda com ele. Quando o time reconhecer que superestimou sua velocidade, ele deve comunicar ao Negócio para rearranjar as histórias dentro de sua nova velocidade. Quando houver problemas com as estimativas, devem-se reavaliar essas estimativas das histórias. É preciso ter coragem para buscar um processo que seja transparente. A comunicação e o feedback são importantes para que os dois jogadores aprendam a melhor forma de maximizar o jogo.

Capítulo 11

Spikes de planejamento

"Algumas vezes, a melhor maneira de resolver um problema é ir para casa, jantar, assistir TV, ir para a cama, e, então, acordar na manhã seguinte e tomar um banho."
– Robert C. Martin

Os spikes de planejamento fazem parte de uma prática que ajuda o time a gerar o conhecimento necessário para estimar as histórias de usuários corretamente, diminuindo, assim, o risco no planejamento [39]. O objetivo é aumentar a confiança para um bom jogo do planejamento. Um spike é um programa muito simples para explorar soluções potenciais [59].

Faça spikes arquiteturais para descobrir respostas em dificuldades técnicas e em problemas de design, ou para conhecer novas tecnologias (APIs, frameworks, ferramentas, linguagens de programação etc.). Quando uma dificuldade técnica importante surgir, deixe um par de desenvolvedores fazer

uma por alguns dias até ter certeza suficiente sobre sua solução.

Por ser uma tarefa a ser realizada, os spikes podem ser estimados. Portanto, trata-se de um momento planejado, garantindo-se que será feito durante uma iteração. Preferencialmente, faça-o em uma iteração e somente desenvolva a história de usuário relativa na próxima. Lembre-se que ele serve para diminuir o risco da estimativa que servirá no próximo planejamento, e também não se esqueça de **não aproveitar** seu código.

11.1 Jogue fora o código gerado no spike

Trate o spike como um experimento e assuma que seu código será jogado fora. Vá direto ao ponto para resolver a incerteza. Ele serve para saber como resolver um problema, e não para produzir um código (de produção) que o resolva [54]. Não tente criar programas úteis ou reutilizáveis em produção. O spike serve para aprendizado, uma vez que se trata de uma prática para descoberta. Por isso, ignore todas as preocupações no entorno da questão.

Vale fazer um paralelo neste ponto. A prototipação é uma técnica de engenharia de software para a descoberta de requisitos com a avaliação do cliente. Normalmente, o protótipo é jogado fora, porque é lento, o código não é funcional e serve apenas para a descoberta, a validação e o refinamento dos requisitos.

Semelhante a essa técnica, o spike é um método para a descoberta de soluções com a avaliação técnica, que, inclusive, também tem seu código jogado fora.

Dica: spike é exceção, não regra

Essa prática é para ser utilizada apenas quando necessária, sendo uma exceção, não uma regra. Toda estimativa de história de usuário possuirá incertezas e riscos; isso é intrínseco da complexidade de desenvolver software.

Capítulo 12

Projeto simples do início ao fim

"Simplicidade é mais complicada do que você pensa. Mas ela é bem valiosa."
– Ron Jeffries

Obter um projeto simples não é uma tarefa fácil; porém, manter a simplicidade agiliza no desenvolvimento de novas funcionalidades e na resposta a mudanças. O projeto simples está presente em todas as fases do XP. A simplicidade é abordada de um modo *extremo*. Ele começa assim e mantém-se por meio de testes automatizados e refatoração.

Tentar prever o futuro é *antiXP*. Nenhuma funcionalidade *adicional* é implementada, pois desvia do caminho da solução e aumenta a complexidade do software. Apenas tenha cuidado com as dívidas técnicas que poderão ser geradas, ou com funcionalidades inacabadas. A programação em pares evita o design extra, porque um membro da dupla não vai querer esperar que o outro desenvolva coisas sem relevância.

Para termos um código limpo, a refatoração é necessária, assim evitando os *code smells* (maus cheiros) nele. Um código é simples [30] quando:

- Roda todos os testes;
- Expressa todas as ideias necessárias;
- Não contém código duplicado;
- Possui a menor quantidade de classes, métodos e variáveis;
- É conciso e legível.

Em todos os níveis de granularidade de software deve-se pensar na simplicidade; ou seja, do código ao produto. Quando o projeto simples atinge o produto, temos o conceito do produto mínimo viável, detalhado a seguir.

12.1 MVP: produto mínimo viável

"Simplicidade a arte de maximizar a quantidade de trabalho não realizado é essencial."
– Décimo princípio do Manifesto Ágil

Ter o projeto simples é essencial para criar um MVP. O MVP (do inglês, *Minimum Product Viable* Produto Mínimo Viável) possui a maior quantidade de funcionalidades de valor na menor entrega possível, sendo suficiente para o cliente utilizar e dar feedback ao desenvolvimento.

A expressão **YAGNI** (do inglês, *You're Not Gonna Need It* Você não precisará disso) lembra todos a desenvolver somente o necessário nas histórias atuais, até no caso de que se saiba que será necessário no futuro.

Menos é mais. A simplicidade tem foco no desenvolvimento da essência do software. Pequenas entregas de valor representam muito do que o cliente necessita. Alguns programadores (não XP) adoram entregar algo a mais ao cliente; já os *extremos* não fazem isso. Programadores XP mantêm o projeto simples do início ao fim.

Dica: MVP é estratégia de entrega

Para trabalhar com MVP é preciso ter uma estratégia na evolução das entregas. Um ótimo exemplo real é o de uma equipe que, na primeira iteração, desenvolveu somente o algoritmo necessário e entregou apenas uma planilha com os resultados que o cliente precisava em seu negócio. Na duas próximas, a equipe complementou as regras nesse algoritmo e desenvolveu a interface com o usuário. O MVP entregue na primeira iteração resolveu a maior parte do problema do cliente! A estratégia para entregar o que restava nas duas próximas iterações completou o trabalho, deixando-o muito satisfeito.

Capítulo 13

Metáfora de sistema

“98% de nosso pensamento está realmente acontecendo em um nível inconsciente, grande parte através de metáforas.”
– Esther Derby

Metáforas em geral estão no nosso cotidiano. No dia a dia, frequentemente as usamos para expressarmos ideias de um modo mais simples, fácil e rápido; tentamos encontrar algo semelhante que a pessoa já conheça para ela conseguir entender o que queremos explicar.

No mundo da computação, fazemos uso de ainda mais metáforas. Você trabalha com documentos, arquivos, pastas e ferramentas em sua *área de trabalho* do computador. Você já deve ter comprado na internet por meio de um *carrinho de compras*. Talvez você já tenha implementado um algoritmo *percorrendo uma árvore em largura ou em profundidade*; ou já tenha resolvido uma *dívida técnica* em seu software. Quem sabe você já enviou um build

para o *pipeline* da integração contínua, ou então participou de um debate em forma de *fishbowl*.

No XP, a metáfora de sistema traz uma visão comum que auxilia o time e o cliente a entender os elementos do sistema. Ela funciona como um *pattern* de alto nível e explica seu design sem uma documentação pesada. Ela é semelhante à linguagem ubíqua, porém, enquanto esta se concentra em uma linguagem comum entre o negócio e os desenvolvedores, aquela foca na linguagem da arquitetura da solução.

O nascimento dessa prática deu-se no sistema de folha de pagamento da Chrysler [9]. A metáfora era de uma fábrica para *produzir* o salário, onde as caixas desciam em uma linha de montagem para serem montadas. As caixas são as partes do salário a serem montadas em um item apenas: o salário do funcionário. Um outro exemplo é o de James Shore em seu primeiro projeto XP, que era de *data warehouse* (armazém de dados) [53], cujos nomes das classes do sistema utilizavam a metáfora de uma empilhadeira em um armazém.

Muitas pessoas ignoram o uso dessa prática por não conhecerem sua utilidade. Uma metáfora de sistema ajuda a padronizar os nomes dos objetos para entendê-lo e também auxilia a reutilizar códigos. Desse modo, o desenvolvedor pode descobrir se algum objeto já existe no software e decidir onde é o melhor local para colocá-lo. Por trazer um vocabulário compartilhado, é uma excelente fonte de comunicação entre a equipe e pode facilitar também na discussão das estimativas. Apenas uma boa metáfora pode ser realmente efetiva.

13.1 Descobrindo uma boa metáfora

Não é fácil encontrar uma metáfora útil para todo sistema [35]. Ela é representada por um domínio bem entendido por todos (domínio fonte), para descrever os objetos e o funcionamento do domínio do sistema (domínio alvo).

Descobrir uma metáfora é uma tarefa criativa; o segredo está em achar um domínio fonte que facilite o entendimento do domínio alvo. Pelo fato de todas serem inexatas, reexamine continuamente a metáfora atual, já que ela pode possuir uma extensão maior ou ser inadequada; esteja pronto para

deixá-la caso isso ocorrer [19].

Joshua Kerievsky define os três Is da metáfora: Iluminação, Inspiração e Integridade [35]. Uma boa metáfora auxilia a iluminar o *design*, inspirar novas ideias e ajudar a sentir que o projeto do sistema está coerente, encaixando-se bem em conjunto. Deve-se buscar uma que contenha esses três itens.

Capítulo 14

Reunião diária em pé

"Ter as melhores ideias, o código mais afinado, ou o pensamento mais pragmático é, em última instância, estéril, a menos que você possa se comunicar com outras pessoas."
– Andy Hunt & Dave Thomas

Você já participou de reuniões longas e improdutivas? A resposta é tão óbvia que até parece uma pergunta boba. Reuniões estão no topo da lista de coisas entediantes que desperdiçam o tempo da maioria dos desenvolvedores.

Para o time no XP, as reuniões diárias em pé do inglês *Daily Stand Up Meeting* são focadas e rápidas, ocorrem no começo do dia e duram até quinze minutos. Seu formato físico é simples: todos de pé formam um círculo e cada pessoa tem a sua vez de falar no sentido horário (ou anti-horário). Ela faz com que o time mantenha-se alinhado, trocando conhecimento constantemente.

Ao começar a fazê-las, diversos erros comuns podem ocorrer. Neste capítulo, traremos um catálogo deles.

14.1 Time alinhado

Cada membro do time responde sucintamente três questões básicas:

- O que eu fiz ontem?
- O que eu farei hoje?
- O que há de problemas no meu caminho?

Dica: vinte segundos por questão e por participante

Uma dica é que cada uma das três questões não deva necessitar mais do que vinte segundos para ser respondida [43]. Desse modo, cada participante não precisa mais que um minuto para falar e um time de dez pessoas levará dez minutos ou menos de reunião.

O propósito da reunião é a comunicação do time inteiro. Em todo começo de dia, ele comunica seus problemas, suas soluções e promove a colaboração. Reuniões curtas com todo o time são mais efetivas do que diversas reuniões com pedaços dele. O planejamento diário sincroniza seu trabalho e suas expectativas, discutindo-se as tarefas realizadas e as tarefas a realizar.

14.2 Troca de conhecimento

Na reunião diária, cada indivíduo sabe o que o outro está trabalhando e se possui problemas relacionados a isso. Qualquer membro do grupo pode pedir e oferecer ajuda ao outro, tanto em dúvidas de negócio, quanto de técnicas. Mesmo que cada programador puxe as tarefas das quais tem conhecimento suficiente para desenvolver, algumas virão somente durante seu desenvolvimento. Muitas vezes, algo que foi dito na reunião servirá como referência no futuro. Uma boa forma para pedir ajuda é parear (programar em pares) com

a pessoa referente ao assunto em questão. Assim, o aprendizado e a troca de conhecimento tornam-se constantes e colaborativos.

14.3 Como começar?

O ideal é começar as diárias desde o primeiro dia da primeira iteração criada no jogo do planejamento. A cultura de experimentação é essencial para começar, visto que, raramente, uma diária sai certa pela primeira vez em um time recém-formado. Por mais que ele seja excelente, deve-se entender que o grupo está em formação e algumas coisas terão de ser adaptadas e melhoradas. Temporariamente, um alarme pode ser usado para ajudar a lembrá-lo do horário do começo e do fim da reunião. Fique atento aos erros comuns listados a seguir e melhore continuamente as reuniões diárias em pé.

14.4 Erros mais comuns de uma reunião em pé

"Ao longo dos anos, eu tenho desenvolvido uma simples regra: quando a reunião fica entediante, saia."
– [ref the-clean-coder

A reunião diária em pé não existe para desperdiçar tempo, mas o contrário; ela substitui outras reuniões para ganhar-se tempo. Porém, é necessário conhecer os erros mais comuns para identificá-los quando ocorrerem. O coach deve ter conhecimento suficiente para ajudar o time a detectá-los e resolvê-los.

Os erros mais comuns em reuniões diárias em pé são:

- **Reunião de status:** a diária não é uma reunião de *status reporting* para uma pessoa, tal como um coordenador ou gerente. Caso seja necessário, a dica é realizá-la após alguém fazer o *status reporting*, possivelmente o *Tracker*.

- **Discutir soluções:** um dos propósitos dessa reunião é identificar problemas, não resolvê-los. A dica é continuar com a reunião diária, deixando a conversa sobre as solução para depois dela.

- **O time não vê valor:** isso pode ocorrer quando um time ágil está se formando. A dica é levantar esse problema na próxima reunião de retrospectiva do time. Não postergue.

- **Não ter um horário fixo:** todos do time precisam saber o horário fixo diário da reunião para segui-lo estritamente. O horário da diária deve ser sagrado. Uma dica é ter o papel do "chato da diária"; uma pessoa eleita para lembrar do horário e dar um toque caso a reunião demore ou perca o foco. Outra dica: um despertador ajuda a lembrar o time.

- **Sem local dedicado:** é importante manter um único lugar para que o time faça a diária, de preferência ao próprio local de trabalho. A dica é fazer a reunião próximo ao quadro de histórias e tarefas para a gestão visual.

- **Não saber ouvir:** para uma boa comunicação, é necessário saber falar e também ouvir atentamente. Uma dica é ter um *token* para chamar a atenção de todos. Ele pode ser qualquer objeto que chame a atenção. Se for um objeto engraçado, fica mais fácil de pedir a atenção de todos. Já vimos equipes utilizando canetas de quadro, apagadores, guarda-chuvas e até garrafa de óleo diesel (em uma empresa de refinaria). *O token é como uma bengala: é para ser temporário até que o time caminhe bem nas diárias.*

- **Não fazer a reunião, porque um indivíduo está ausente:** é melhor ter uma reunião sem um membro do time do que não tê-la. A dica é simples: apenas faça a reunião.

- **Não ficar de pé:** uma dica de produtividade é fazer com que todos os participantes fiquem de pé na reunião para que ela seja rápida e tenha foco. Isso faz com que as pessoas prestem mais atenção nas outras. Reuniões improdutivas com elas sentadas fazem com que seja confortável a participação. Caso ela perca o foco, as pessoas ficarão cansadas e logo vão querer dar um jeito de realinhar e finalizá-la.

Lembrete!

A comunicação é o maior propósito do time XP na reunião em pé diária.

Capítulo 15

Posse coletiva

"Myyy PRECIOUSSS."
– Gollum, no filme O Senhor do Anéis

Posse coletiva é uma prática indispensável no XP, pois envolve colaboração, comunicação e feedback. Com a posse coletiva, qualquer membro ou par do time pode implementar uma funcionalidade, arrumar um bug ou refatorar em qualquer parte do código, a qualquer momento. No XP, todos têm a responsabilidade pelo sistema. Isso encoraja cada membro do time a sentir-se responsável pela qualidade do todo. É claro que nem todos saberão sobre todas as partes igualmente, mas todos saberão, ao menos, um pouco de cada parte do sistema.

O termo original no livro do eXtreme Programming [6] é *collective ownership* e não apenas *code ownership*, porque todos os artefatos no desenvolvimento são de posse coletiva do time, não só o código. Além dele, a suíte de

testes é também de propriedade coletiva, assim como o ambiente de integração contínua e os executáveis do build.

Quando um par do time encontra uma oportunidade de melhorar o código, ele pode tranquilamente refatorar e melhorar seu *design*. A posse coletiva gera bastante sinergia com as outras práticas do XP, e o versionamento com a integração contínua avisará a todos o que está sendo alterado.

O trabalho em time com a posse coletiva encoraja a difusão de ideias e de conhecimento. A programação em par ajuda a distribuir esse conhecimento por meio do aprendizado entre os membros do grupo, deixando a posse coletiva mais refinada. Porém, para tudo isso funcionar, é necessário que nenhum integrante ache que é o dono de partes do sistema. É preciso colaboração.

15.1 My precious!

Você assistiu ao filme *O Senhor dos Anéis*? O personagem Gollum (ou Sméagol) chamava o anel de *my precious* (em português, "meu precioso"). Não faça isso com o *seu* código. Deixe seu time colaborar! O objetivo é evitar as ilhas de conhecimento, reduzindo o risco de depender apenas de uma pessoa. A posse coletiva é um benefício para todos os membros (cada um poderá sair de férias ou tirar uma folga com mais tranquilidade por não dependerem tanto dele, por exemplo).

Quando a posse não é coletiva, somente uma pessoa é dona do código e os programadores devem submetê-lo para revisão e aprovação. Isso é, basicamente, o oitavo desperdício no Lean [41], que é relativo ao capital humano e à criatividade dos funcionários. Consequentemente, isso gerará diversos outros desperdícios: espera (o time aguardará a revisão do dono do código); transporte (enviar o código ao dono); defeito (nem toda a decisão do dono será a melhor); superprocessamento (a revisão para controle pelo dono do código); e estoque (a fila de itens a revisar pelo dono).

Uma pessoa é notavelmente um gargalo no time ou no processo quando não consegue atender todas suas demandas e cria estoque de tarefas, nas quais ninguém pode ajudar. E atenção: qualquer membro poderá sair da empresa a qualquer momento, o que pode impactar no negócio do cliente.

Reflexão!

Faça uma reflexão: quantas pessoas são necessárias faltar ao trabalho para que o sistema não possa mais ser desenvolvido? Quanto menor o número, pior será a situação. A posse coletiva aumenta esse número ao máximo, ou seja, ao tamanho da equipe.

Capítulo 16

Padrão de codificação

"A quantia de tempo gasto lendo código versus escrevendo é bem mais de 10 para 1. Então, fazer o código mais fácil de ler, o torna mais fácil de escrever."
– Robert C. Martin

Pelo fato de todos no time XP estarem trabalhando juntos em cada parte do sistema, refatorando código e trocando de pares frequentemente, não se pode haver diferentes formas de codificação. Um padrão torna-se necessário, pois deve facilitar a comunicação entre o time, encorajando a posse coletiva, e evitando problemas na programação em pares e na refatoração.

O padrão deve ser estabelecido e concordado pelo time, pois faz com que todos o utilizem efetivamente. Caso contrário, a equipe ficará insatisfeita e evitará ao máximo seu uso. Sua existência vale muito mais do que a sua forma, porque ele pode ser revisto e evoluído de acordo com seu uso.

Algumas linguagens de programação já propõem padrões de codificação, e também é comum uma organização ou um produto estabelecê-los. Na linguagem Java, a Sun estabeleceu em 1997 o *Java Code Conventions* (Convenções de Código Java) [46], válido até os dias atuais. A Google define seus próprios guias de estilos para diversas linguagens, como: C++, Java, Objective-C, Python, Shell, HTML/CSS, JavaScript, e Lisp [29]. O kernel do Linux também define seu estilo de codificação [58].

Imagine centenas ou milhares de programadores produzindo código para um único software sem padronização? Seria o caos! O kernel do Linux provavelmente teria tantos *bugs* que se tornaria inutilizado. O Free Software Foundation também define padrões de codificação para seus programas GNU [21], descrevendo o melhor uso da linguagem C, padrões para interfaces, para linha de comando, para documentação e para as *releases*.

Ok, esses são projetos grandes. Mas e os projetos pequenos? É o que discutiremos logo a seguir.

16.1 Pequenos projetos também se beneficiam?

Certamente, pequenos projetos com times pequenos se beneficiarão com padrões de codificação. Ok, alguém ainda pode não ter se convencido e pensar que isso é necessário apenas para grandes projetos. Veja a seguir como um pequeno código pode causar problemas.

Falta de padronização de código

Sem uma padronização, um programador pode fazer tranquilamente qualquer um dos quatro códigos mostrados na sequência para calcular o módulo de um número na variável `a` (o péssimo nome é proposital não temos um padrão nesse momento). Esses quatro exemplos vêm da combinação de duas variações: ter uma condição `if` com o bloco interno de código na mesma linha ou não, e ter chaves ou não.

Condição com bloco interno na mesma linha:

- *Com chaves:*

```
if (a <= 0) { a = -1 * a; }
```

- *Sem chaves:*

```
if (a <= 0) a = -1 * a;
```

Condição com bloco interno na próxima linha:

- *Com chaves:*

```
if (a <= 0) {
  a = -1 * a;
}
```

- *Sem chaves:*

```
if (a <= 0)
  a = -1 * a;
```

Ok, sem muitos problemas até então, pois fizemos apenas duas variações nesse pequeno trecho de código. Porém, e se variarmos em mais formas? Se as chaves serão indentadas ou não, se terá `else`, se usaremos nomes de variáveis sem significado (letra `a`) ou não. Com isso, teremos mais 3 tipos de variações, totalizando 5 tipos em 32 formas de codificação. Além disso, certamente existem outras mais. Programadores têm suas preferências de estilos ao programar, e essa variabilidade tenderá ao infinito. Então, sem uma padronização de código, haverá retrabalho, discussão e estresse, mesmo em um pequeno projeto.

16.2 Como definir?

Antes de tudo, tente começar com algo que já exista. Pesquise se há um padrão de codificação na sua empresa ou se há algum definido pela linguagem ou tecnologia utilizada. Caso positivo, comece por isso, permitindo que o time revise e aprimore-o. Caso não exista, reúna o grupo inteiro para defini-lo. Ninguém melhor que ele próprio para padronizar seu próprio código.

Para descrever um padrão de código, diversos itens podem ser considerados:

- Nomenclatura de variáveis;
- Nomenclatura de métodos;
- Nomenclatura de classes;
- Nomenclatura de pacotes;
- Indentação da tabulação;
- Indentação de chaves;
- Indentação das estruturas condicionais;
- Uso de parênteses em expressões;
- Uso de letras maiúsculas e minúsculas;
- Uso de `else` no `if`;
- Uso de `default` no `switch`;
- Uso de tratamento de exceções;
- Uso de estruturas de dados específicas;
- Importação de classes e bibliotecas;
- Diretrizes de Orientação a Objetos;
- Uso de padrões de projeto;
- Boas práticas de programação;
- Comentários de código;
- Comentários nos *commits*;
- *Logging*;
- Detalhes específicos da linguagem de programação;
- Detalhes específicos do ambiente de desenvolvimento.

Capítulo 17

Programação em par

"Dados olhos suficientes, todos os erros são óbvios."
– Lei de Linus, Eric Steven Raymond

Sabe-se que revisão por pares (*peer review*) é uma boa prática de desenvolvimento que aumenta a qualidade do software e a colaboração no time. Visto que é produtivo, que tal programar e revisar em pares o tempo inteiro?

No XP, todo o código de produção é criado por programação em par (*pair programming*). Isso significa ter duas pessoas piloto e copiloto trabalhando juntas em apenas um computador, com foco em uma única tarefa e ao mesmo tempo. O que o XP indica é programar em par quando o código de produção for escrito. Não necessariamente se deve programar em par 100% do tempo, pois atividades, como pesquisas e leituras, não têm essa necessidade.

Os nomes dos papéis de um par vêm da metáfora da aviação: piloto e copiloto (ou navegador). O piloto programa, enquanto o copiloto acompa-

nha e cuida de tudo o que está sendo realizado, pensando em casos de testes, desde o padrão de codificação até questões de arquitetura. A troca de papéis é essencial para colaboração e partilha de conhecimento.

Trabalhar dia a dia ao lado de um colega exige habilidades sociais que levam tempo em aprender, porém aumenta a cooperação no time, independente do status da função de cada integrante. Programação em par não é mentoria, uma vez que é um trabalho colaborativo entre ambas as partes, independente de terem muita diferença nas experiências. Não é aceitável dizer *O design que você fez possui um erro*, mas sim *O design que nós fizemos possui um erro*.

Nas seções deste capítulo, falaremos sobre os diversos benefícios desse tipo de programação, assim como discutiremos o que gestores podem pensar sobre seu custo. Veremos também sobre a pressão do par durante o desenvolvimento, o nivelamento do conhecimento entre a dupla e como começar (com dicas sobre pareamento e o ambiente de trabalho). Além disso, pergunto: por que não trabalhar em par em todos os tipos de tarefas? :)

17.1 Diversos benefícios

Diversos benefícios são conhecidos no *pareamento* [62]:

- **Revisão de código contínua:** muitos erros são pegos quando eles estão sendo codificados, em vez de serem descobertos por um testador, tornando a quantidade de erros consideravelmente menor;
- **Discussão contínua da solução:** o design é aprimorado e o código fica mais sucinto;
- **Afinamento do par:** o time resolve problemas mais rapidamente por estar coeso também em pares;
- **Aprendizagem:** as pessoas aprendem significativamente sobre o sistema e sobre engenharia de software;
- **Gestão do conhecimento:** o projeto acaba com várias pessoas entendendo cada pedaço do sistema;

- **Trabalho colaborativo:** os indivíduos aprendem a trabalhar em time e a conversar mais frequentemente, dando mais fluxo à informação e à dinâmica do grupo;
- **Motivação:** as pessoas apreciam mais o seu trabalho e o trabalho de todos.

17.2 Um desenvolvedor pelo preço de dois?

O quê? Dois programadores fazendo o trabalho de um? De modo algum! Isso é o que é dito geralmente por quem não conhece a produtividade e os benefícios da programação em par, ou por pessoas que ainda pensam no processo fordista de ter-se apenas um funcionário por estação de trabalho. Produzir softwares é uma atividade complexa e não se deve aplicar processos fabris. O coach XP deve ficar atento à organização do ambiente, tratar da mudança com a visão gerencial e harmonizar os relacionamentos humanos.

1 + 1 > 2. Duas pessoas trabalhando em um único computador produzirá mais que duas pessoas trabalhando separadas, já que isso aumenta o foco, e também gera qualidade e troca de conhecimento. Com um breve tempo, as vantagens já surgirão, basta acreditar no potencial da equipe. Quem pratica sente seus benefícios.

17.3 A pressão do par

Você já ouviu falar em pressão do time? Isso acontece quando se trabalha com somente uma meta. As pessoas dependem de você, e você delas. Caso alguém não colabore na meta, o próprio time vai naturalmente pressioná-la; ou a própria pessoa se cobrará de alguma forma.

No trabalho em par, isso não é diferente. Cada membro tem o compromisso de fazer um bom trabalho. Isso gera maior concentração, incentiva o bom trabalho e aumenta a responsabilidade. Revezar frequentemente faz com que cada um respeite o outro, tratando a pressão como algo bom e não como um problema. *Jogue limpo com o seu par, você dependerá dele.*

17.4 Nivelando o conhecimento

O aprendizado pela troca de conhecimento está na essência da programação em pares. O programador que souber menos logo aprenderá com o mais experiente, podendo aprender uma tecnologia, um framework, um algoritmo, um padrão de projeto, uma história de usuário, ou qualquer componente do desenvolvimento de software. O nivelamento de conhecimento é um resultado intrínseco do *pareamento*.

17.5 Como começar?

O segredo para começar é ter um clima de experimentação, entendendo que algumas coisas não funcionarão de início e precisarão de melhoria. No jogo do planejamento, histórias e tarefas podem ser elencadas para serem desenvolvidas em pares. Na retrospectiva, o time reflete sobre os benefícios trazidos e o que deve ser melhorado no *pareamento*.

Uma boa dica para fazer acontecer esse trabalho em par é limitar o desenvolvimento a um número de tarefas menor que o de programadores. Assim, sempre haverá, ao menos, um programador impedido de pegar uma tarefa, então ele precisará parear com um outro. Utilizando um quadro de Kanban, basta aplicar o limite no *WIP* (do inglês, *Work in Progress* Trabalho em Progresso).

Dica: revise em pares quando não parear

Mesmo que algumas atividades não sejam realizadas em pares o tempo inteiro, utilize sempre a revisão em dupla. O modo mais eficiente é o de rever o código dessa forma **antes** de enviá-lo para o *branch* padrão. Isso resulta um código totalmente revisado que será integrado no seu servidor. Reexaminar depois pode perder a prioridade e gerar retrabalho. Aproveite também para usar a revisão não apenas no código, mas também em tudo o que é útil para o desenvolvimento.

17.6 Dicas para aprimorar a programação em par

Que tal experimentar essas abordagens?

- **Vamos tentar sua ideia primeiro:** alguém da dupla sugere começar pela ideia de solução do outro integrante. Essa abordagem cria um clima de respeito por entender que as soluções de cada um serão experimentadas e melhoradas.

- **Pensar alto:** em vez de o piloto ficar olhando para o código e pensando sozinho, ele fala o que está pensando. Isso ajuda o copiloto a entender o que está sendo feito.

- **Regra dos dez segundos:** muitas vezes o piloto está em um raciocínio aprimorado e não linear. Por essa razão, o copiloto deve respeitar esse momento e aguardar cerca de dez segundos para intervir. Essa dica é muito boa para não quebrar o raciocínio.

- **Revezar por ciclo de tempo:** revezar o papel de piloto e copiloto é essencial. Para forçar a troca, o par pode colocar um alarme para despertar no final do ciclo de tempo. Esse ciclo não pode ser nem muito curto, nem muito grande. O par saberá o tempo certo, que normalmente varia de dez minutos a uma hora. A Técnica Pomodoro é uma forma eficiente de revezar o par, controlar o tempo e ter foco. O par desenvolve o software a cada vinte e cinco minutos, e faz um intervalo curto de três a cinco minutos. A cada *pomodoro*, o copiloto assume o lugar do piloto e vice-versa. A cada quatro *pomodoros*, o intervalo é mais longo, de vinte a trinta minutos.

- **Revezar por ciclo de TDD:** revezar por tempo pode quebrar o raciocínio do piloto por parar no meio da escrita. Revezar por ciclo de TDD (vide capítulo 19) torna interessante a linha de raciocínio seguida. Uma bela variação é que o copiloto assuma na fase de codificação [12], e não na de escrita de testes, fazendo com que um escreva o teste para o outro programar e refatorar.

17.7 O ambiente de trabalho

Trabalhar fisicamente em pares não exige muita infraestrutura, sendo suficiente apenas um computador configurado normalmente. Para que os pares trabalhem juntos a distância ou sem um espaço físico adequado, existem diversas ferramentas de trabalho remoto com compartilhamento de tela e de acesso remoto de áudio e vídeo.

Cuide do espaço físico. A mobília pode dificultar ficar lado a lado programando (por exemplo, em mesas de canto ou em formato de 'L'). Os programadores terão dores no corpo depois de um dia de programação em pares com pouca ergonomia. Uma dica é adaptar o espaço; melhor ainda se houver estações de programação em par, adicionando um teclado, um mouse e um monitor o copiloto, o que é mais favorável para a ergonomia do trabalho.

17.8 Especificar, projetar, trabalhar, tudo em par

Mesmo que essa prática do XP chame-se *programação em pares*, o trabalho como um todo pode ser realizado em dupla. Um programador pode parear com o cliente para tirar dúvidas; o cliente pode parear com um testador para escrever bons testes de aceitação; e um programador pode parear com um testador para discutir alguns testes unitários.

Dica: pareamento em tudo!

Somos tão adeptos do trabalho em par que pareamos em atividades além do desenvolvimento de software. Lecionados cursos em par, palestramos em eventos em par etc. Sabemos que, assim, um curso ou uma palestra tem muito mais qualidade, motivação e troca de conhecimento. Então, que tal começar a parear também?

Capítulo 18

Refatoração de código para alta qualidade

"Qualquer tolo pode escrever código que o computador possa entender. Bons programadores escrevem código que humanos possam entender."
– Martin Fowler

Você se lembra das aulas de matemática sobre fatoração? O objetivo era aprender como fatorar uma fração ou equação matemática. Por exemplo, simplificar a expressão `2*x+2*y` para sua expressão fatorada `2*(x+y)`.

Pois bem, a refatoração tem o mesmo propósito na codificação. Com origem na Orientação a Objetos, é também chamada de *refinamento incremental* e mantém a semântica da funcionalidade, alterando apenas o *design*.

Uma refatoração é uma mudança feita na estrutura interna do software que o faz ficar mais fácil de entender e mais barato de modificá-lo, sem mu-

dar seu comportamento observável [23]. Obter essa simplicidade não é algo fácil, por isso a fazemos por meio de uma série de pequenas alterações, uma a uma sem alterar esse comportamento. Atenção: refatoração não é um ato de reescrever código.

Deve-se pedir permissão para refatorar? Acreditamos que não. A qualidade é importante no software, e você é o profissional que decide sobre o uso da refatoração. Por essa razão, ela está embutida na estimativa do desenvolvimento. Refatorar é um investimento na qualidade do código e não pode ser visto como retrabalho.

Quais são seus benefícios?

- Economiza tempo e aumenta a qualidade;
- Melhora o design do software;
- O código torna-se mais legível e reutilizável;
- Fica mais fácil encontrar defeitos;
- Ajuda você a programar mais rápido.

A abordagem do XP é refatorar impiedosamente; ou seja, não se opta por isso. Falaremos disso a seguir, bem como sobre *bad smells* de código.

Dica: papel do cleaner

Tenha alguém no time XP com o papel do *cleaner*. Isso auxilia o time a criar a cultura de refatoração.

18.1 Refatore impiedosamente

Refatorar não é uma tarefa opcional. Escrever código já na primeira vez com um excelente design é difícil. Quando você pensar em refatoração, criará estratégias para a codificação e melhorias no software. Pensando no time, a posse coletiva de código (capítulo 15) é importante para que qualquer indivíduo sinta-se confortável para refatorar.

Não deixe para refatorar apenas quando se utiliza TDD, pois existem outros momentos em que você pode fazê-lo, como:

- Ao adicionar uma função;
- Ao consertar um defeito;
- Para facilitar o entendimento;
- Em revisões de código;
- Regra de três: na primeira vez que você fizer algo, apenas faça. Na segunda vez, a duplicação surgirá, mas mantenha assim. Na terceira vez, refatore.

Dica: pense no próximo!

Escreva o código para pessoas, não apenas para o compilador. Seu time agradecerá e alguém no futuro certamente ficará feliz em ver um bom código estruturado.

18.2 Bad smells de código

Sabemos que alguns tipos de código são difíceis para alterar, como: código com lógica duplicada ou complexa, código ilegível ou código com comportamento adicional e desnecessário. Estes comumente possuem muitos *bad smells* (maus cheiros), também chamados *code smells* (cheiros de código). Podemos dizer que se podem descobrir rapidamente os *bad smells* com uma breve revisão do código.

Os maus cheiros podem estar nos níveis de classe, de método ou de atributo. A refatoração é a remoção da duplicação e de *code smells*. Duplicação é sinal de código sujo, de que há mais código do que o necessário em relação a classes, métodos e variáveis. Obter o código limpo (*clean code*) é o objetivo da refatoração, pois ele fica legível, com alta coesão, baixo acoplamento e sem maus cheiros.

Alguns exemplos de *code smells* a serem refatorados são:

- Uma classe ou método muito longo;
- Uma classe que utiliza muitos métodos de outra classe;
- Uma classe que faz muito pouco;
- Subclasses muito semelhantes;
- Nome de variável não comunicativo;
- O não tratamento de uma exceção;
- Código duplicado;
- Código inútil.

Uncle Bob (Robert C. Martin) lista em seu livro, *Código limpo* [42], diversos *odores*; pérolas; heurísticas sobre problemas de comentários e de ambiente de build; e testes gerais de nomenclatura e de testes. Vale a pena a leitura; é muito recomendado para quem deseja fazer um código realmente bom.

Dica: catálogos de refatorações

Martin Fowler mantém na internet um catálogo de refatorações bastante completo e detalhado. Nele, você encontrará refatorações, como: adicionar um parâmetro, extrair uma superclasse, mover um método, trocar um array por um objeto e muito mais. Confira-o em [26].

Além disso, Joshua Kerievsky disponibiliza em seu livro, *Refatoração para padrões*, uma lista bastante detalhada de como refatorar para padrões [34].

Capítulo 19

TDD: Desenvolvimento Guiado por Testes

"Eu não sou um excelente programador; eu sou apenas um bom programador com excelentes hábitos."
– Kent Beck

O TDD (do inglês *Test-driven Development* Desenvolvimento Guiado por Testes) é uma técnica para construção de software que guia seu desenvolvimento por meio da escrita de testes. Sua essência está em seu *mantra*: testar, codificar e refatorar (veja na imagem a seguir). Essa técnica surgiu a partir da refatoração, dando base para guiar a codificação. TDD gera aprendizado, porque ajuda a aprender boas práticas de programação e a criar testes automatizados.
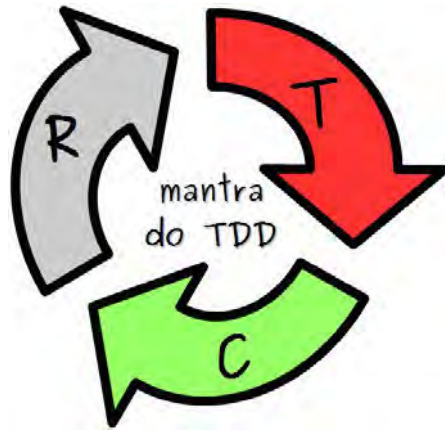
Fig. 19.1: Mantra do TDD: testar, codificar e refatorar

Mas espere um pouco! Escrever testes antes de codificar? Sim, exatamente! No TDD, primeiramente é escrito o código de teste e depois (somente depois) o de produção. Dessa forma, a codificação será guiada por testes e o código de produção estará coberto por eles. Os testes guiarão o desenvolvimento. É uma mudança de paradigma que vale a pena conhecer.

Joshua Kerievsky traz a visão desse mantra como um diálogo [34]:

- **Pergunta:** você pergunta ao sistema por meio de um teste.
- **Resposta:** você responde escrevendo código que passa no teste.
- **Refinamento:** você refina a pergunta consolidando ideias, removendo itens desnecessários e clarificando ambiguidades.
- **Repetição:** você mantém o diálogo fazendo novas perguntas.

Na agilidade, trabalhamos com ciclos de trabalho em todos os níveis do projeto. No nível do negócio, utilizamos ciclos de entregas e de iterações para o desenvolvimento de software; no nível do trabalho em equipe, trabalhamos com ciclos em reuniões diárias; e no da codificação, o TDD traz ciclos de trabalho para desenvolver as funcionalidades incrementalmente. Cada um

dura alguns minutos: adiciona-se um teste, fazendo-o funcionar com código de produção e refatorando-o.

Sobre os benefícios do TDD, Maurício Aniche fez um compilado de resultados em pesquisas sobre isso [4]. Eles são bastante convincentes no uso de TDD:

- Programadores produziram código que passaram em aproximadamente 50% mais testes caixa-preta do que os que não utilizavam;
- Redução de 40-50% na quantidade de defeitos e um impacto mínimo na produtividade;
- 87,5% dos programadores acreditam que facilitou entender os requisitos;
- 95,8% acreditam que TDD reduziu o tempo gasto com *debug*;
- 78% acreditam que aumentou a produtividade da equipe;
- 92% acreditam que ajuda a manter um código de maior qualidade;
- 79% acreditam que promove um design mais simples.

19.1 Padrões de TDD

Para detalhar o TDD, Kent Beck [8] define quatro classes de padrões: padrões gerais de TDD, padrões de teste para TDD, padrões de barra vermelha, e padrões de barra verde. Vamos apresentá-las ao longo deste capítulo.

Padrões gerais de TDD:

- Teste seu software com testes automatizados;
- Um teste não deve afetar a execução do outro;
- Mantenha uma lista de testes;
- Escreva um teste antes de escrever código de produção;
- Comece a escrita do teste pelo método `assert(...);`;

- Os dados de testes devem ter significados evidentes.

Padrões de teste para TDD:

- Quando escrever casos de teste grandes, comece com sua menor parte que faça quebrar;
- Quando testar recursos caros ou complicados, utilize constantes que simulem o comportamento desses recursos (com estratégias de *dummy*, *stub*, *spy*, *fake* e *mock*);
- Quando estiver programando sozinho, deixe o último teste quebrado. Quando você voltar a programar, ele dirá onde você parou.;
- Quando estiver programando em time, deixe todos os testes rodando. Seu grupo precisará ter todos os testes rodando.

Os três passos do **mantra do TDD** são descritos a seguir.

Passo 1: escrever um teste que falhe

Feedback ao programador é um ponto-chave no XP e é guiado por bons testes automatizados. Aí está a importância de automatizá-los: ter um feedback, uma resposta rápida para guiar o desenvolvimento do software. Sua automatização é um investimento a curto e longo prazo, pois guia a codificação e aumenta a qualidade do software a longo prazo.

Uma dúvida bastante comum é se o TDD faz uso apenas de testes de unidade. A resposta é negativa, porque ele é ortogonal em relação a ele. Estes verificam unidades de comportamento. O TDD utiliza, principalmente, testes de unidade, mas também pode usar testes funcionais e de aceitação para guiar o desenvolvimento das funcionalidades.

Para facilitar sua escrita são utilizados frameworks xUnit: JUnit (Java), NUnit (.NET), CPPUnit (C++), PHPUnit (PHP), e mais uma grande diversidade dessas ferramentas. As expressões "barra vermelha" (*Red bar*) e "barra verde" (*Green bar*) surgiram desses frameworks, pois são avisos visuais em uma barra colorida integrada no ambiente de desenvolvimento. A vermelha

é o resultado de execução de um ou mais testes falhando; já a verde é o resultado da execução de todos os testes passando.

Para escrever um teste que falhe (barra vermelha), Kent Beck define os padrões:

- Escreva um teste que gere aprendizado e que se esteja confiante para implementar;
- Inicie com um teste para uma operação que não faz nada;
- Peça explicações e esclarecimentos por meio dos testes;
- Escreva testes para aprender o que um software externo faz;
- Anote os outros testes que surgirem para não perder o foco do atual;
- Quando houver um defeito encontrado, escreva um teste para apontar o *bug*.

Passo 2: escrever código funcional para rodar o teste

Agora, com um teste já falhando, é hora de escrever o código de produção para fazê-lo passar (barra verde). Dessa forma, o código de produção ficará praticamente todo coberto por testes. Neste segundo passo, o objetivo é fazer o código passar, sem se preocupar tanto com o seu design, porque isso será feito na próxima etapa: a refatoração.

Ao usarmos *baby steps* (passos de bebê), o código é produzido até dar *barra verde*. Rode os testes constantemente. Os *baby steps* ajudarão a verificar cada passo, você terá feedback contínuo deles. Entregue frequentemente no repositório de código (algumas vezes ao dia). Todos eles devem estar rodando para enviar o código ao repositório; a integração contínua garantirá isso.

Padrões de barra verde para TDD, para fazer rodar o teste quebrado:

- Retorne uma constante para simular o resultado e, então, transforme-a gradualmente em uma expressão com variáveis;
- Transforme os retornos constantes em expressões, após possuir dois ou três exemplos;

- Implemente diretamente as operações simples;
- Quando a implementação usar coleções de objetos, inicie sem coleções e, somente depois, utilize-as.

Dica: utilize seus baby steps conscientemente

Em seu livro sobre TDD, Maurício Aniche traz a dica de que os *baby steps* devem ser realizados de maneira consciente: o desenvolvedor deve buscar pela solução mais simples (e não necessariamente pela modificação mais simples) [4]. Assim como um bebê, o desenvolvedor deve dar um passo seguro, de acordo com a complexidade do problema que está sendo resolvido.

Passo 3: refatorar o código

"O objetivo do TDD é escrever código limpo que funciona."
– Ron Jeffries

O propósito do TDD está no design do código; o teste é apenas um efeito colateral (ótimo, por sinal). Agora com o código coberto pelo teste, é hora de melhorar seu design, refatorando-o. Um dos erros mais comuns no uso de TDD está em não refatorar constantemente. Tenha muita atenção nessa parte; se você não a fizer, terá um código mais complexo e não fará TDD.

Segundo Martin Fowler, um dos gurus da refatoração, esta é uma mudança feita na estrutura interna do software para deixá-lo mais fácil de entender e barato de modificar, sem mudar seu comportamento observável [23]. Uma das mágicas do TDD está aqui: o comportamento observável é guiado pelo feedback dos testes! Eles apontarão se a refatoração do código o alterou ou não.

Cada refatoração deve ser realizada em *baby steps*; ou seja, não se deve fazer grandes refatorações. O feedback contínuo é essencial para evitar que o comportamento observável não seja alterado. Caso seja, os testes automatizados apontarão o problema. Um *baby step* deve ser simples, mas não simplório.

Assim como em um passo de bebê, você deve dar o maior passo para ir em frente com total segurança.

Lembre-se de refatorar também seus testes. Eles fazem parte do software, e se não forem refatorados, se tornarão complexos e dificultarão a manutenção futura.

19.2 Show me the code

"Talk is cheap. Show me the code."
– Linus Torvalds

Falar é fácil, mostre-me o código. Ok, vamos começar com um exemplo clássico praticado em *Coding Dojos*: a conversão de números decimais para romanos. Para isso, escolhemos a linguagem Java e o JUnit para escrever os testes de unidade. Como primeiro ciclo de TDD, vamos converter o número decimal `1` para romano. Segue o código do teste:

```
import static org.junit.Assert.*;
import org.junit.Test;
public class RomanNumberConverterTest {
    @Test
    public void testI() {
        assertEquals("I", RomanNumberConverter.converter(1));
    }
}
```

Esse teste falhará a própria compilação do código, pois inexiste a classe `RomanNumberConverter`, e muito menos seu método `converter`. Uma dica é utilizar a IDE *Integrated Development Environment* (Eclipse, por exemplo) para criar as classes e métodos automaticamente, a partir dos erros de compilação. Precisamos fazer esse teste compilar com o código a seguir:

```
public class RomanNumberConverter {
    public static Object converter(int i) {
        return null;
    }
}
```

Veja que o código está com uma má aparência, mas já compila e o teste já pode ser executado. Porém, ele falha, pois retorna `null` e deveria retornar `I`. Visto isso, faremos o primeiro teste funcionar:

```
public class RomanNumberConverter {
    public static Object converter(int i) {
        return "I";
    }
}
```

Nosso primeiro teste passou! Você pode estar achando estranho retornar uma constante, mas calma! Lembre-se dos *baby steps*, além da simplicidade no XP. Como regra, deve-se retornar uma constante para fazer funcionar os primeiros testes, apenas depois abstrair em variáveis em uma fórmula.

Agora, possuímos teste passando, portanto podemos refatorar. Faremos as seguintes refatorações, também com *baby steps*:

- Tornar mais sugestivo o nome do método de teste `testI` para `deveRetornarI`;
- Mudar o retorno do método `converter` de `Object` para `String`;
- Tornar o método `converter` de classe para um método de objeto (removendo o `static`).

Eis o código com essas refatorações aplicadas:

```
public class RomanNumberConverterTest {
    @Test
    public void deveRetornarI() {
        assertEquals("I",
            new RomanNumberConverter().converter(1));
    }
}
public class RomanNumberConverter {
    public String converter(int i) {
        return "I";
    }
}
```

Lembre-se de rodá-lo a cada pequena refatoração. Desse modo, teremos garantido que não o quebramos.

Fechamos o nosso primeiro ciclo do mantra do TDD: criamos um teste, fazemo-lo funcionar e o refatoramos em *baby steps*. Estamos aprendendo o problema. Agora, é o momento de irmos para o nosso segundo ciclo de TDD. Hey, atenção! Não saia escrevendo o código funcional! Você precisa de um teste falhando antes de tudo. Ok, então vamos escrever o próximo:

```
public class RomanNumberConverterTest {
    @Test
    public void deveRetornarI() {
        assertEquals("I",
            new RomanNumberConverter().converter(1));
    }
    @Test
    public void deveRetornarII() {
        assertEquals("II",
            new RomanNumberConverter().converter(2));
    }
}
```

Ah, agora sim temos um teste falhando, então podemos escrever código funcional:

```
public class RomanNumberConverter {
    public String converter(int i) {
        if (i == 1) {
            return "I";
        }
        else return "II";
    }
}
```

Esse código está ruim, porém os testes estão rodando. Esses são os *baby steps*. O XP foca no necessário e somente o necessário. Agora é hora de refatorar, melhorando seu *design*. Vamos fazer uma pequena refaforação, tornando o nome da variável do método `converter` de `i` para `decimal`. Resultando no código:

```
public class RomanNumberConverter {
    public String converter(int decimal) {
        if (decimal == 1) {
            return "I";
        }
        else return "II";
    }
}
```

Acabamos nosso segundo ciclo de TDD. Vamos para o terceiro. Agora você já sabe: primeiro escreva um teste que falhe. Vamos escrever um teste para verificar a conversão do decimal três:

```
public class RomanNumberConverterTest {
    @Test
    public void deveRetornarI() {
        assertEquals("I",
            new RomanNumberConverter().converter(1));
    }
    @Test
    public void deveRetornarII() {
        assertEquals("II",
            new RomanNumberConverter().converter(2));
    }
    @Test
    public void deveRetornarIII() {
        assertEquals("III",
            new RomanNumberConverter().converter(3));
    }
}
```

Ok, temos o teste falhando para o valor três. Vamos agora criar código funcional. Só que dessa vez, realizaremos o *baby step* um pouco maior, começando o algoritmo efetivo da solução. Faremos uso da recursividade:

```
public class RomanNumberConverter {
    public String converter(int decimal) {
        if (decimal == 0) {
            return "";
```

```
        }
        return converter(decimal - 1) + "I";
    }
}
```

Os três testes estão passando, excelente. Vamos refatorar? Que tal colocarmos um `else` com chaves nessa condição `if`?

```
public class RomanNumberConverter {
    public String converter(int decimal) {
        if (decimal == 0) {
            return "";
        } else {
            return converter(decimal - 1) + "I";
        }
    }
}
```

Ótimo! Testes passando e código refatorado. Temos o terceiro ciclo de TDD completo.

Para não ficar exaustivo esse exemplo, vamos adiantar alguns ciclos de TDD. Fizemos mais um ciclo de TDD para cada um dos números decimais: 5, 6, 7, 8, 9, 10, 28 e 34. O código de testes resultante desses ciclos ficou dessa forma:

```
public class RomanNumberConverterTest {
    @Test
    public void deveRetornarI() {
        assertEquals("I",
            new RomanNumberConverter().converter(1));
    }
    @Test
    public void deveRetornarII() {
        assertEquals("II",
            new RomanNumberConverter().converter(2));
    }
    @Test
    public void deveRetornarIII() {
        assertEquals("III",
```

```
            new RomanNumberConverter().converter(3));
    }
    @Test
    public void deveRetornarIV() {
        assertEquals("IV",
            new RomanNumberConverter().converter(4));
    }
    @Test
    public void testDeveRetornarV() {
        assertEquals("V",
            new RomanNumberConverter().converter(5));
    }
    @Test
    public void testDeveRetornarVI() {
        assertEquals("VI",
            new RomanNumberConverter().converter(6));
    }
    @Test
    public void testDeveRetornarVII() {
        assertEquals("VII",
            new RomanNumberConverter().converter(7));
    }
    @Test
    public void testDeveRetornarVIII() {
        assertEquals("VIII",
            new RomanNumberConverter().converter(8));
    }
    @Test
    public void testDeveRetornarIX() {
        assertEquals("IX",
        new RomanNumberConverter().converter(9));
    }
    @Test
    public void testDeveRetornarX() {
        assertEquals("X",
            new RomanNumberConverter().converter(10));
    }
    @Test
    public void testDeveRetornarXXVIII() {
```

```
        assertEquals("XXVIII",
            new RomanNumberConverter().converter(28));
    }
    @Test
    public void testDeveRetornarXXXIV() {
        assertEquals("XXXIV",
            new RomanNumberConverter().converter(34));
    }
}
```

O código funcional resultante ficou assim:

```
public class RomanNumberConverter {
    public String converter(int decimal) {
        if (decimal == 0) {
            return "";
        } else if (decimal <= 3) {
            return this.converter(decimal - 1) + "I";
        } else if (decimal == 4) {
            return "IV";
        } else if (decimal <= 8) {
            return "V" + this.converter(decimal - 5);
        } else if (decimal == 9) {
            return "IX";
        } else if (decimal <= 39) {
            return "X" + this.converter(decimal - 10);
        }
        return null;
    }
}
```

Você deve perceber que a funcionalidade ainda não está pronta e o código não está totalmente limpo. Podemos ainda fazer algumas boas refatorações, como: mudar o retorno de `null` para uma `String` vazia ou criar uma enumeração para os números romanos.

Ainda precisamos fazer mais testes. Precisamos verificar casos de erro (quando o decimal for menor que zero), além de verificar as conversões de números maiores com as letras de cinquenta (L), cem (C), quinhentos (D) e

mil (M). Sugerimos que você pratique TDD com esse exercício e, de preferência, que complete todos os casos de teste.

Dica: pratique em um Coding Dojo!

Fazer uma sessão de *Coding Dojo* propicia um excelente espaço aberto e momento de aprendizado para treinar a programação em pares, TDD, projeto simples e outras práticas do XP. Daniel Wildt tem um post que explica mais sobre isso em [60].

Capítulo 20

Integração contínua

“Integração contínua não livra os bugs, mas os tornam dramaticamente mais fáceis de encontrar e de remover.”
– Martin Fowler

Integração contínua (IC) é uma prática, na qual o código que está sendo desenvolvido pelo time é integrado, versionado, construído e verificado diversas vezes ao dia em um ambiente dedicado. Os programadores XP devem integrar e fazer *commit* em somente uma versão no repositório de código. Cada integração é verificada por um *build* com testes automatizados, detectando erros o mais cedo possível [24].

Essa prática gera sinergia com as outras do XP. Os *builds* das pequenas entregas são construídos por meio da IC. Ela auxilia na refatoração, pois verificará o *build* automaticamente a cada *commit* realizado no repositório. Em cada um, o padrão de codificação pode ser verificado automaticamente.

A integração contínua é um processo simples que traz muitos benefícios:

- Aumenta o feedback, a comunicação na equipe e a moral do time;
- Todos veem o que está acontecendo;
- Previne e descobre os problemas de integração mais cedo;
- Reduz riscos e evita a baixa qualidade;
- Não é necessário um integrador dedicado para a equipe;
- Todos têm acesso à versão mais atualizada;
- Auxilia na reutilização de código: os desenvolvedores terão sempre o código mais atualizado;
- Evita problemas de *merge*, fazendo pequenas integrações ao longo do tempo.

Para ter tais benefícios de um modo eficiente, é preciso seguir algumas regras que potencializam a integração contínua. Além disso, é necessário a utilização de ferramentas para as tarefas automatizadas relativas. Para detalhar melhor, falaremos dessas regras e de ferramentas nas próximas seções deste capítulo.

20.1 Como potencializar os benefícios?

"Quanto maior for a razão aparente para criar um branch, mais você não deveria criar um branch."
– Jez Humble

Um ambiente de integração contínua precisa estar bem afinado para que gere excelentes benefícios. Para isso, trazemos um compilado de dicas:

- Tenha um servidor de integração dedicado;
- Tenha apenas uma única fonte de repositório de código;

- O time faz os *commits* no *branch* padrão (*head*, *trunk*, *master*);
- Não quebre o *build*, toda mudança deve manter o código rodando;
- Antes de fazer commit, todos os testes de unidade devem rodar e passar no build em seu ambiente de desenvolvimento local;
- Arrume imediatamente os *builds* quebrados;
- Automatize o *build* e mantenha-o rápido;
- Os testes automatizados devem ser rodados com sucesso no *build*, especialmente os testes de aceitação automatizados;
- Torne fácil o acesso do último *build* a todos do time;
- Crie penalidades leves e descontraídas para quem quebrar os *builds*;
- Automatize o deploy;
- Teste em um ambiente clone do ambiente de produção;
- Não desative a integração contínua quando estiver sob pressão, nessa hora ela terá ainda mais valor.

Você acha complicado para um time sempre usar integração contínua? Pois saiba que o Google e o Facebook mantêm todo o seu desenvolvimento apenas no *trunk*! É uma questão de disciplina e atitude. O *trunk* é o principal *branch*, gerando o nome *Trunk Based Development* (Desenvolvimento Baseado em Trunk) a essa abordagem.

No Google, em 2010, já havia mais de 5 mil desenvolvedores trabalhando no mesmo repositório diariamente, em mais de 40 escritórios, com mais de 20 alterações por minuto, com 50% da base de código sendo alterada mensalmente, e mais de 50 milhões de casos de testes executados diariamente [38]. No caso do Facebook, ele possui três *trunks*: um para o *www* (web), outro para Android (mobile) e outro para iOS (mobile). Portanto, é bem acessível que um time apenas consiga utilizar integração contínua, certo?

"Vai ter que pagar bala!"

Quando alguém da equipe for o responsável por ter quebrado o *build*, ele deve comprar um pacote de doces para preencher o pote de balas do time. Essa é uma maneira leve e descontraída de evitar que ele seja quebrado por descuidos. O maior problema pode ocorrer quando o pote está cheio e o time não aguenta mais comê-las. ;)

20.2 Ferramentas

Antes de falarmos de ferramentas, é preciso estar claro que integração contínua é muito mais uma atitude do que apenas o uso de ferramentas. Ferramentas servem para dar suporte a um processo; ou seja, uma por si só não faz nada. Algumas das mais utilizadas para essa integração atualmente são: o Jenkins (*fork* do Hudson), o Bamboo e o CruiseControl. Essas ferramentas trabalham em conjunto com as de versionamento, de *build*, de qualidade de código e de testes automatizados. Há uma gama delas para cada tecnologia.

As ferramentas de versionamento controlam o código-fonte, gerenciando as versões do software. Existem dois modelos principais:

- **Modelo cliente-servidor:** todas as versões do código ficam centralizadas no servidor. Os clientes possuirão apenas a versão de trabalho e executarão comandos sobre o repositório central. Exemplos dessas ferramentas são o Subversion (SVN) e o Concurrent Version System (CVS);

- **Modelo distribuído:** uma cópia integral de todo o repositório de código fica em cada cliente, ou seja, cada um também é um servidor, não sendo necessário ter um central. Exemplos famosos desse modelo de ferramentas são o Git e o Mercurial.

As ferramentas de *build* empacotam os executáveis do software de acordo com as configurações e uma série de tarefas, também gerenciando dependências de bibliotecas. Normalmente, as ferramentas de desenvolvimento (IDEs)

já têm integração a essas de *build*, sendo um processo transparente para o desenvolvedor. Exemplos desse tipo são o Ant, o Maven e o MSBuild.

As ferramentas de qualidade de código preocupam-se com o seu design. Muitos problemas podem ser automaticamente detectados por elas. São os principais tipos:

- Analisadores de código (por exemplo, o FindBugs e o PMD);
- Verificadores de cobertura de código (por exemplo, o JaCoCo e o Code Coverage);
- Verificador de padrão de codificação (CheckStyle);
- Frameworks de testes de código (JUnit e PHPUnit);
- Gerenciador de qualidade de código (Sonar).

Sobre ferramentas de teste, existem diversas para cada tipo. Para testes funcionais na web, existe o Selenium; para os de desempenho e de carga em Java, o JMeter; de integração em Java, o Arquillian; e de cenários e critérios de aceitação, o Cucumber, o JBehave (Java) e o SpecFlow (.NET).

Quando o *build* falhar, a ferramenta de IC notificará todos sobre o ocorrido. O Jenkins possui um *plugin* bem-humorado em que o Chuck Norris aparece em uma imagem com uma *cara de quem não gostou* nem um pouco do *build* com falha, ou *feliz da vida* quando ele foi realizado com sucesso. Veja nas imagens a seguir:

Fig. 20.1: Chuck Norris não gostando nem um pouco que o build falhou. Fonte da imagem: https://wiki.jenkins-ci.org

Fig. 20.2: Chuck Norris contente que o build foi construído com sucesso. Fonte da imagem: https://wiki.jenkins-ci.org

Esse *plugin* pode ser encontrado em https://wiki.jenkins-ci.org/display/JENKINS/ChuckNorris+Plugin.

Capítulo 21

Ritmo sustentável

"Desenvolvimento de software é uma maratona, não uma sprint."
– Robert C. Martin

Imagine uma pessoa correndo em uma maratona. O caminho é longo, então a melhor estratégia é correr em um ritmo constante até a chegada. Todos os corredores profissionais sabem disso. Caso alguém queira correr mais rápido que seu ritmo, em poucos minutos ele estará ofegante e terá que parar para descansar, também perdendo sua concentração na respiração. O mesmo ocorre ao desenvolver software.

Ter ritmo sustentável é uma regra criada pelo XP para balancear o desenvolvimento com as demandas do negócio. Todos têm seus ritmos equilibrados: o time XP, o cliente e pessoas envolvidas com o projeto. Utilizar a simplicidade traz soluções mais efetivas ao cliente, mais ainda quando há mais demandas do negócio do que a capacidade do time. A qualidade do software

é essencial para manter esse fluxo de trabalho normalizado, porque defeitos gerariam retrabalho para as próximas iterações.

Produtividade em longo prazo é o ponto-chave aqui. Durante uma semana ou pouco mais, o time consegue produzir mais trabalhando com horas extras. Porém, a longo prazo é inviável, porque o rendimento vai decaindo semana a semana.

Dica: cultura Lean

A dica é buscar a cultura Lean em seu desenvolvimento de software. No Lean, a sobrecarga é chamada de *Muri*, que é gerada pelo desbalanceamento de carga (chamado de *Mura*), que, por sua vez, gera diversos tipos de desperdícios (chamados de *Muda*) [48]. A sobrecarga gera cansaço, que gera desatenção, que gera defeitos, que gera retrabalho: um ciclo que só vai diminuindo a produtividade e fazendo com que o time possa abandonar outras práticas do XP.

Planejar o trabalho, de acordo com a velocidade do grupo e em um ritmo de trabalho sem horas extras, auxilia no ritmo sustentável. Vamos falar disso nas seções deste capítulo.

21.1 Velocidade do time

"Os processos ágeis promovem desenvolvimento sustentável. Os patrocinadores, desenvolvedores e usuários devem ser capazes de manter um ritmo constante indefinidamente."
– Oitavo princípio do Manifesto Ágil

Um bom jogo do planejamento ajuda no ritmo sustentável, uma vez que será o time que dará as estimativas e negociará as entregas com o cliente. Sua velocidade é importante para determinar um planejamento realista, sem desbalanceamento. Para o grupo encontrar sua velocidade, são necessárias algumas iterações. No XP, ela é dada em *story points*. Existe uma variação grande sobre estimativas, mas o que realmente interessa é que o time a conheça para estabelecer um ritmo sustentável.

É papel do time proteger-se de trabalho extra. Algo curioso pode acontecer com grupos que estão em um excelente ritmo sustentável com entregas de qualidade: a gestão pode ver que ele está indo muito bem e tentar *empurrar* mais tarefas, o que pode comprometer o restante do trabalho e começar a produzir entregas atrasadas de baixa qualidade.

Horas extras é um sintoma gerado por sérios problemas em um projeto [6]. Muitas vezes, a solução mais rápida para um gestor é fazer com que seus funcionários trabalhem mais, em vez de arrumar o problema raiz. Isso vai contra a melhoria contínua, acoberta o problema real e gera novos.

Dica: planeje e estime todas as tarefas necessárias

Faça um planejamento de todas as atividades necessárias com suas estimativas. Planeje todas as tarefas necessárias para implementar as histórias de usuário, com o objetivo de finalizar uma iteração ou uma entrega com todo o trabalho pronto. Além das tarefas de codificação, planeje e estime as de integração, de teste (incluindo automatizados) e de deploy, como também outras importantes. Caso sobrem atividades para a próxima iteração, negocie e replaneje o trabalho.

21.2 40 horas semanais

"Eu quero estar revigorado e cheio de energia toda manhã, e cansado e satisfeito toda noite."
– Kent Beck

O conceito de ritmo sustentável iniciou-se com a prática de 40 horas semanais (*40-hour week*) no livro de origem do *eXtreme Programming* [6]. Ron Jeffries abstraiu esse conceito e o renomeou para *ritmo sustentável* [5]. Curiosamente, há várias décadas, em 1926, Ford já havia estabelecido a jornada de trabalho de 40 horas semanais, pois sabia que o problema seria resolvido com uma boa organização, e não com mais horas trabalhadas [37]. A imagem a seguir ilustra a relação da produtividade por semana de trabalho em um ritmo sustentável (40 horas semanais) e insustentável (60 horas semanais, fazendo horas extras).
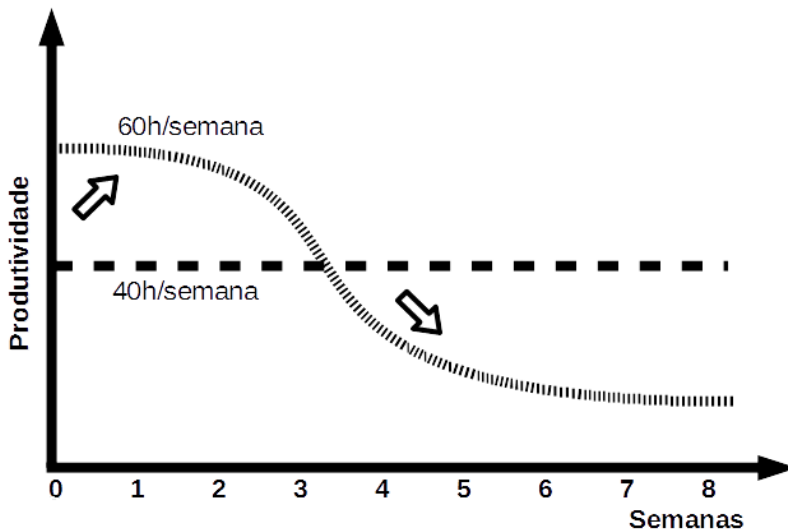
Fig. 21.1: Gráfico de produtividade por semana para o trabalho de 40 horas semanais e para o com horas-extras de 60 horas

Pessoas não são máquinas, e desenvolver software não é uma atividade simples. É necessário raciocínio aprimorado para produzir cada funcionalidade que é diferente da anterior já feita. Programadores não são *mão de obra*, mas sim trabalhadores do conhecimento (chamados de *cérebro de obra*). Sabe-se que, nos humanos, o cérebro é o órgão que consome mais energia, totalizando até 20% do total da energia do corpo [57]. Comumente, um dia de trabalho rende pelo trabalho normal de uma semana, assim como o inverso; uma semana inteira não rende nem por um dia normal de trabalho.

Algumas pessoas têm capacidade de trabalhar em todo seu potencial criativo, atento e confiante em 45 horas por semana, outras em 35 horas, mas raramente alguém conseguirá isso em 60 horas, em longo prazo. Um time cansado trabalhará menos, não importa quanto tempo trabalhem a mais.

Capítulo 22

Indo além do eXtreme programming

"É o que você aprende, depois de você saber tudo, o que importa."
– John Wooden

No capítulo 2, já falamos sobre *Lean Software Development* e *Scrum*, que são complementares ao XP. A ideia deste capítulo é trazer brevemente algumas das principais abordagens e práticas para ir além do eXtreme Programming. Certamente, há muitas outras, mas já é um bom começo para aprender ou reforçar. Reflita com seu time sobre como esses assuntos poderão auxiliar no desenvolvimento.

22.1 Jogos e dinâmicas

Que tal discutir e aprender as práticas do XP de um modo divertido? Com jogos e dinâmicas, seu time pode trocar muito conhecimento sobre esse método. Na Industrial Logic, em 2001, Joshua Kerievsky desenvolveu cartas de baralho para aprender XP, além de uma lista de jogos com elas [33]. O baralho possui cartas **V** que representam os valores do XP; **P** que retratam os problemas no desenvolvimento de software; e **S** que são as soluções trazidas pelas práticas do XP (vide imagem a seguir).

Fig. 22.1: Extreme Programming playing cards

Em 2014, com a permissão da Industrial Logic, Jamile Alves e Dionatan Moura traduziram para português e as adaptaram para serem disponibilizadas em um formato livre para impressão [3]. Assim, você poderá usá-las com seu time. Experimente os jogos de acordo com seu contexto e lembre-se de se divertir! :-)

22.2 Behaviour-Driven Development (BDD)

O *Behaviour-Driven Development* (BDD Desenvolvimento Guiado por Comportamento) possui o mesmo propósito do XP: unir as pessoas de desenvolvimento com as de negócio. O BDD baseou-se no TDD [47], e serve também para análise ágil e teste de aceitação automatizado. Ele utiliza a especificação por exemplos [1], uma abordagem colaborativa para definir requisitos a partir de exemplos realistas, em vez de enunciados abstratos. Cada exemplo é entendido por meio de um cenário com dados reais. Uma história de usuário pode possuir diversos exemplos reais de utilização. Um de cenário poderia ser:

- **Cenário:** pesquisando *extreme programming* no Google.
- **Dado que** estou na pesquisa do Google
- **E** digito *extreme programming* no campo de busca
- **Quando** pressiono a tecla Enter
- **Então** é exibida a primeira página de uma lista dos websites com conteúdo relevante sobre o assunto.

Quando o software possuir essas especificações executáveis, ele terá a **documentação viva**; ou seja, ela estará conectada diretamente ao código por meio dos testes de aceitação. Ferramentas de frameworks auxiliam a escrita e a execução dos cenários JBehave (Java), Cucumber (Ruby), RSpec (Ruby), SpecFlow (.Net), e Jasmine (JavaScript).

22.3 Domain-Driven Design (DDD)

Domain-Driven Design (DDD Projeto Orientado ao Domínio) é uma abordagem criada por Eric Evans que conecta a implementação do software a um modelo, em um contexto específico, por meio de uma linguagem ubíqua [19]. Ela coloca o foco primário do projeto no domínio, além de permitir a colaboração entre as pessoas técnicas e as especialistas nele, tudo isso em um

refinamento iterativo. No XP, o DDD é complementar às metáforas de sistema e auxilia a comunicação do time para entender melhor sobre o negócio do cliente.

22.4 Kanban

Kanban significa "cartão visual" em japonês. Na linha de produção de uma fábrica, um kanban acompanha peças ou partes específicas para sinalizar visualmente a entrega de uma determinada quantidade de itens, agilizando o processo.

No desenvolvimento de software, o sistema kanban auxilia na visualização do trabalho em andamento e de cada passo na cadeia de valor (da demanda do cliente até estar em produção). Ele também permite limitar o trabalho em progresso (WIP *Work in Progress*), removendo gargalos na produção e dando vazão ao trabalho. Ele embasa as decisões no processo, visualiza sua consequência e identifica oportunidade de melhorias no fluxo da produção. Tudo isso dando base para a produção puxada (trabalhar com a demanda atual existente, e não com base em previsões).

No XP, as histórias de usuário e tarefas vão para o quadro (escritas em cartões ou post-its) e são gerenciadas visualmente. Em termos do Lean, o uso desse sistema catalisa o pensamento Lean na entrega de sistemas de software. Para aprender mais sobre ele, sugerimos o livro *Kanban em 10 Passos*, de Jesper Boeg [10].

22.5 Estimando com planning poker

Para estimar as histórias de usuário e conversar sobre elas durante o jogo do planejamento, você e seu time poderão utilizar o *planning poker*. É uma técnica de dimensionamento e planejamento das histórias por meio de um consenso do time. Para isso, cada integrante usará um baralho específico e pontuará a história; todos farão isso ao mesmo tempo, como em um poker. O grupo deve fazer rodadas de pontuação até chegar em um consenso. O ponto-chave está na comunicação.

O baralho é baseado na sequência de Fibonacci; possui as cartas de `0, 1, 2, 3, 5, 8, 13, 20, 40` e `100` (vide imagem a seguir). A razão

do uso dessa sequência exponencial é relacionada à incerteza porque, quanto maior for a pontuação, maior será a incerteza da estimativa. A unidade de estimativa é a que o time definiu (*story points*, dias ideais, tamanhos de camiseta etc.). Para estimar cada história, o grupo deve conversar sobre seus detalhes e o que será necessário para desenvolvê-la. Lembre-se que estimativas são literalmente estimativas: uma aproximação, e não um prazo formal.

Fig. 22.2: Cartas do baralho de planning poker

22.6 Resolvendo dívidas técnicas

Suponha que você precise implementar uma funcionalidade em seu sistema e você possui duas opções: a primeira é a de entregar um código bem feito e refatorado, com um bom *design*, mas que tomará um pouco a mais de tempo; e a segunda é a de entregar rapidamente um código confuso no qual você espera nunca mais tocar, porque as mudanças ali serão cada vez mais complicadas.

A primeira opção poderá gerar algumas (raras) dívidas técnicas, porém a segunda certamente causará muitas. O XP defende a primeira opção, mas é possível que se comece a usá-lo em um sistema já existente. Por isso, ressaltamos as dívidas técnicas.

Dívida é dívida, e de algum modo você terá que pagar. Pensando por um lado ruim, você pagará apenas os juros (cada vez mais altos), que resultam em vários *bugs*, assim como atrasos e custos maiores, entrega a entrega. E a dívida não para por aí: seu código ficará cada vez pior, seu time com mais estresse físico e mental, seu cliente insatisfeito, algumas pessoas se demitirão, ou até o sistema parará de vez.

Por um lado pior ainda, elas ficarão tão grandes que o trabalho será apenas resolver *bugs*. Já pensando por um lado positivo, você utilizará a refatoração para ir arrumando continuamente as maiores dívidas técnicas, automatizando testes e resgatando o ritmo sustentável do time, assim como a satisfação do cliente.

22.7 Refatorando também o banco de dados

Refatoração no XP é uma disciplina que se aplica a todo o sistema, inclusive ao banco de dados. Você já pensou no impacto que uma base de dados com um péssimo *design* tem? Esse tipo de refatoração pode ser definida como uma mudança disciplinada na estrutura de uma base de dados que não altera sua semântica, porém melhora seu projeto e minimiza a introdução de dados inconsistentes [44].

O princípio de *code smells* também é aplicado a banco de dados, chamado de *database smell* (mau cheiro de banco de dados). Fabrízio Mello possui um material sobre *database smells* em [45]. Essa área de refatoração em banco é recente e possui poucos materiais, mas que já são bastante úteis para começar a refatorar o seu.

22.8 Código limpo

Ok, já falamos de código limpo (*clean code*) nos capítulos 12, 19 e 18, mas a importância é tão grande no XP que vale a pena reforçar essa prática. O livro *Código limpo* [42] do Uncle Bob (Robert C. Martin) apresenta a disciplina de escrever código limpo, trazendo diversas dicas, como: nomes significativos, funções, comentários, formatação, objetos, estruturas de dados, tratamento de exceções, testes de unidade, classes, concorrência e odores de código. E lembre-se sempre: código limpo não é o que mais se refatora, mas o que menos se suja.

22.9 Entrega contínua e DevOps

Entrega contínua é um conjunto de princípios e práticas com o objetivo de compilar, testar e liberar software ao cliente de forma mais rápida e frequente.

Essa abordagem encaixa-se muito bem com as práticas do XP de liberação constante de pequenas versões e de integração contínua. Podemos dizer que entrega contínua também é uma prática *extrema*, pois está bastante alinhada com o primeiro princípio do Manifesto Ágil: *nossa maior prioridade é satisfazer o cliente por meio da entrega contínua e adiantada de software com valor agregado*.

Jez Humble e David Farley descrevem os princípios da entrega de software no livro *Entrega contínua* [20]:

- Criar um processo de confiabilidade e repetitividade de entrega de versão;
- Automatize quase tudo;
- Mantenha tudo sob o controle de versão;
- Se é difícil, faça com mais frequência e amenize o sofrimento;
- A qualidade deve estar presente desde o início;
- "Pronto" quer dizer "entregue";
- Todos são responsáveis pelo processo de entrega;
- Melhoria contínua.

Seguindo tais princípios, você terá um processo aprimorado de entrega contínua.

22.10 Leituras sobre desenvolvimento de software

Ao longo dos capítulos, referenciamos diversos livros relacionados a cada assunto do XP. Vá além, lendo e pesquisando esses e outros livros. Procure também séries de livros e editoras especializadas em software. Lê-los é uma grandiosa fonte de aprendizado. Leia bastante e desperte em si a curiosidade em aprender mais e mais.

No Brasil, temos a satisfação de possuir uma editora com foco em livros de desenvolvimento de software nas área de Agile, Java, desenvolvimento Web, Mobile, Games e Front-End. Essa editora é a Casa do Código [15]. Seus livros são escritos por autores com renome na comunidade e são revisados por uma curadoria com experiência na área. Com certeza eles o ajudarão a ir muito além do XP.

Lá existem livros na área de metodologia e tecnologia. Na de metodologia, existem livros sobre agile, startup, Scrum, TDD (com Java, .NET, Ruby), DevOps, testes de Software; e na de tecnologia, livros sobre UX, jQuery, HTML5 e CSS3, web design responsivo, games em HTML, JavaScript, Android e IOs, Java 8, JSF, JPA, CDI, REST, Node.js, MongoDB, SQL, PHP, MySQL e muito mais. Show! Não é? :-)

Um outro livro que é um excelente compilado da agilidade escrito por diversos agilistas brasileiros, referências no país e no mundo, é o *Metodos ágeis para desenvolvimento de software* [18]. Ele traz os assuntos sobre o Manifesto Ágil, a história dos métodos ágeis no Brasil, Scrum, XP, OpenUP, FDD (*Feature-Driven Development*), Lean, Kanban, modelagem ágil, DDD (*Domain-Driven Design*), TDD, estimativas, gestão visual, coaching e facilitação de times ágeis.

22.11 Artesanato de Software

Comumente, aprendemos que desenvolver software é uma atividade baseada na engenharia, na qual os desenvolvedores o constroem com métodos, técnicas e ferramentas fundamentadas na engenharia de software e no gerenciamento de projetos. Existe uma lacuna nisso, pois desenvolvê-los não é uma tarefa simples que possa ser resolvida com métodos tradicionais de engenharia. Desenvolvê-los é uma tarefa do conhecimento e complexa.

O movimento do Artesanato de Software (do inglês, *Software Craftsmanship*) reconhece que a produção de software é uma atividade artesanal, e não fabril. Dentro dessa área, o aprendiz deve estudar técnicas, tecnologias e ferramentas para construir bem o código, aplicando bons princípios de programação. E não para por aí: ele deve também criar hábitos de trabalho e investir em suas habilidades sociais (*soft skills*). Participar de uma comu-

nidade também aprimora seu aprendizado. Desse modo, ele se tornará um artesão de software.

Há um grande elo entre um programador XP e um artesão de software. Precisamos de melhores programadores, e não mais simplórios. *Pense em quantas dívidas técnicas um programador simplório pode deixar no sistema ao longo de um ano! É um grande impacto! Provavelmente serão necessários dois ou mais para arrumar toda a bagunça.*

Da mesma forma que métodos ágeis, o Artesanato de Software possui um manifesto [64]. O Edson Yanaga, artesão de software, fez sua tradução [16]:

Manifesto pelo Artesanato de Software: como aspirantes a artesãos de software, elevamos o nível do desenvolvimento de software profissional ao praticar e auxiliar outros a aprender o ofício. Por meio desse trabalho, passamos a valorizar:

- Não somente software funcionando, mas também software bem feito;
- Não somente responder a mudanças, mas também continuamente adicionar valor;
- Não somente indivíduos e interações, mas também uma comunidade de profissionais;
- Não somente colaboração com o cliente, mas também parcerias produtivas.

Ou seja, na busca pelos itens à esquerda descobrimos que os itens à direita são indispensáveis.

Relacionado à arte do Artesanato de Software, temos a parte comportamental, de atitude, que representa a coragem e o respeito como dois dos cinco valores do XP. Adicionalmente a tais valores, Chad Fowler escreveu o livro *O programador apaixonado* [22]; e Uncle Bob (Robert C. Martin), o livro *Codificador limpo* (*Clean coder*) [43]. O primeiro traz diversas dicas para construir uma carreira notável em desenvolvimento de software; já o segundo traz um código de conduta para profissionais de software, sendo pragmático com conflitos, prazos apertados, gerentes pouco razoáveis, dizer não e lidar com a pressão.

Índice Remissivo

Referências Bibliográficas

[1] Gojko Adzic. *Specification by example: how successful teams deliver the right software*. Manning Publications, 2011.

[2] Dionatan Moura; Jamile Alves. Big picture do extreme programming. https://github.com/dsmoura/xp-big-picture, 2015.

[3] Dionatan Moura; Jamile Alves. Xp playing cards: aprendendo extreme programming com muita diversão! http://dionatanmoura.com/2015/05/06/xp-playing-cards/, 2015.

[4] Maurício Aniche. *Test-Driven Development: teste e design no mundo real*. Casa do Código, 2012.

[5] Christoph Baudson. What is sustainable pace? http://www.sustainablepace.net/what-is-sustainable-pace, 2012.

[6] Kent Beck. *Extreme Programming: embrace change*. Addison-Wesley, 1999.

[7] Kent Beck. *eXtreme Programming explained: embrace change. 2. ed.* Addison-Wesley, 2004.

[8] Kent Beck. *TDD Desenvolvimento Guiado por Testes*. Bookman, 2010.

[9] Rilla Khaled; James Noble; Robert Biddle. System metaphor in "extreme programming": a semiotic approach. 2004.

[10] Jesper Boeg. Kanban em 10 passos. http://www.infoq.com/br/minibooks/priming-kanban-jesper-boeg, 2012.

[11] Paulo Caroli; Tainã Caetano. *Fun retrospectives: activities and ideas for making agile retrospectives more engaging*. Lean Publishing, 2014.

[12] Paulo Caroli. *Thoughtworks antologia Brasil: histórias de aprendizado e inovação*. Casa do Código, 2014.

[13] Mike Cohn. *User stories applied: for Agile software development*. Addison-Wesley Professional, 2004.

[14] Lisa Crispin. *Agile testing: a practical guide for testers and agile teams*. Addison-Wesley Professional, 2009.

[15] Casa do Código. Casa do código: livros e tecnologia. http://www.casadocodigo.com.br/, 2015.

[16] Signatários do manifesto. Manifesto pelo artesanato de software. http://www.yanaga.com.br/2011/12/manifesto-pelo-artesanato-de-software.html, 2011.

[17] Kent Beck; et al. Manifesto para desenvolvimento ágil de software. http://www.agilemanifesto.org/iso/ptbr/, 2001.

[18] Rafael Prikladnicki; Renato Willi; Fabiano Milani; et al. *Metodos ágeis para desenvolvimento de software*. Bookman, 2014.

[19] Eric Evans. *Domain Driven Design: atacando as complexidades no coração do software*. Alta Books, 2010.

[20] Jez Humble; David Farley. *Entrega contínua*. Bookman, 2014.

[21] Free Software Foundation. Gnu coding standards. https://www.gnu.org/prep/standards/, 2014.

[22] Chad Fowler. *O programador apaixonado: construindo uma carreira notável em desenvolvimento de software*. Casa do Código, 2014.

[23] Martin Fowler. *Refatoração: aperfeiçoando o projeto de código existente*. Bookman, 2004.

[24] Martin Fowler. Continuous integration. http://www.martinfowler.com/articles/continuousIntegration.html, 2006.

[25] Martin Fowler. Flaccidscrum. http://martinfowler.com/bliki/FlaccidScrum.html, 2009.

[26] Martin Fowler. Catalog of refactorings. http://refactoring.com/catalog/, 2013.

[27] Martin Fowler. Storypoint. http://martinfowler.com/bliki/StoryPoint.html, 2013.

[28] Jr. Frederick P. Brooks. No silver bullet: essence and accident in software engineering. null, 1987.

[29] Google. google-styleguide. https://code.google.com/p/google-styleguide/, 2015.

[30] Ron Jeffries. *Extreme Programming installed*. Addison-Wesley Professional, 2000.

[31] Ron Jeffries. Essential xp: card, conversation, confirmation. http://ronjeffries.com/xprog/articles/expcardconversationconfirmation/, 2001.

[32] Ron Jeffries. Context, my foot! http://ronjeffries.com/xprog/blog/context-my-foot/, 2009.

[33] Joshua Kerievsky. Extreme programming playing cards. http://www.industriallogic.com/blog/xp-playing-cards/, 2001.

[34] Joshua Kerievsky. *Refatoração para padrões*. Bookman, 2008.

[35] Joshua Kerievsky. Interview: Joshua kerievsky on system metaphor. http://www.infoq.com/interviews/kerievsky-metaphor, 2009.

[36] Joshua Kerievsky. Stop using story points. http://www.industriallogic.com/blog/stop-using-story-points/, 2012.

[37] Ashish Kumar. Ford factory workers get 40-hour week. http://www.history.com/this-day-in-history/ford-factory-workers-get-40-hour-week, 2009.

[38] Ashish Kumar. Development at the speed and scale of google. http://www.infoq.com/presentations/Development-at-Google, 2011.

[39] Daniel Wildt; Guilherme Lacerda. Conhecendo o extreme programming (xp). https://codingbyexample.wordpress.com/artigos/, 2010.

[40] Esther Derby; Diana Larsen. *Agile retrospectives: making good teams great*. Pragmatic Bookshelf, 2006.

[41] Jeffrey K. Liker. *O modelo Toyota: 14 princípios de gestão do maior fabricante do mundo*. Bookman, 2005.

[42] Robert C. Martin. *Código limpo: habilidades práticas do Agile software*. Alta Books, 2011.

[43] Robert C. Martin. *O codificador limpo: um código de conduta para programadores profissionais*. Alta Books, 2012.

[44] Fabrízio Mello. Database refactoring. https://codingbyexample.wordpress.com/2013/07/29/database-refactoring/, 2013.

[45] Fabrízio Mello. Bad smells em bancos de dados. http://www.slideshare.net/fabriziomello/xpconf-42476328, 2014.

[46] Sun Microsystems. Development at the speed and scale of google. http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html, 1997.

[47] Dan North. Introducing bdd. http://dannorth.net/introducing-bdd/, 2006.

[48] Roman Pichler. The three m's: the lean triad. http://www.infoq.com/articles/lean-muda-muri-mura, 2008.

[49] Roman Pichler. 10 tips for writing good user stories. http://www.romanpichler.com/blog/10-tips-writing-good-user-stories/, 2010.

[50] Mary Poppendieck; Tom Poppendieck. *Implementando o desenvolvimento Lean de software*. Bookman, 2011.

[51] Jeff Sutherland; Ken Schwaber. Scrum guides. http://www.scrumguides.org/, 2015.

[52] James Shore. The decline and fall of agile. http://www.jamesshore.com/Blog/The-Decline-and-Fall-of-Agile.html, 2008.

[53] James Shore. That funky metaphor stuff. http://www.jamesshore.com/Blog/That-Funky-Metaphor-Stuff.html, 2008.

[54] James Shore. The art of agile development: Spike solutions. http://www.jamesshore.com/Agile-Book/spike_solutions.html, 2010.

[55] Jeff Sutherland. I am jeff sutherland, the co-creator of scrum. ask me anything! http://www.reddit.com/r/IAmA/comments/2hwo5i/i_am_jeff_sutherland_the_cocreator_of_scrum_ask/, 2014.

[56] Ron Jeffries; Jeff Sutherland. A two day deep dive into agile: Scrum, extreme programming(xp) and lean steps to software development success. http://www.gbcacm.org/deep/deepagile2007/, 2007.

[57] Nikhil Swaminathan. Why does the brain need so much power? http://www.scientificamerican.com/article/why-does-the-brain-need-s/, 2008.

[58] Linus Torvalds. Linux kernel coding style. https://www.kernel.org/doc/Documentation/CodingStyle.

[59] Don Wells. Extreme programming: a gentle introduction. http://www.extremeprogramming.org/, 1999.

[60] Daniel Wildt. Quer praticar programação? coding dojo! http://blog.danielwildt.com/2013/06/06/quer-praticar-programacao-coding-dojo/, 2013.

[61] Rafael Helm; Daniel Wildt. Histórias de usuário. http://www.historiasdeusuario.com.br/, 2014.

[62] Alistair Cockburn; Laurie Williams. The costs and benefits of pair programming. 2001.

[63] Klaus Wuestefeld. L and c: learning and coolness. http://klauswuestefeld.blogspot.com.br/2010/07/l-learning-and-coolness.html, 2010.

[64] Edson Yanaga. Manifesto for software craftsmanship. http://manifesto.softwarecraftsmanship.org/, 2009.

[65] Edward Yourdon. *Death march*. Prentice Hall, 2003.